
# Hardware Attacks against Hash-based Cryptographic Algorithms

Aymeric Genet

School of Computer and Communication Sciences

Master Thesis

August 2017

**Responsible**
Prof. Arjen K. Lenstra
EPFL / LACAL

**Supervisor**
Dr. Hervé Pelletier
Nagra Kudelski Group

# Abstract

This thesis surveys the current state of the art of hash-based cryptography with a view to finding vulnerabilities related to side-channel attacks and fault attacks. For side-channel investigation, we analyzed the power consumption of an Arduino Due microcontroller running a custom ARM implementation of SPHINCS-256—the most advanced digital signature scheme based on hash functions. Simple power analysis (SPA) was applied on a single trace to obtain a first insight into the implementation, and then on multiple traces to identify an initial data dependence of the power consumption on the hash functions involved in the instance. Based on this result, differential power analysis (DPA), with difference of means, V-test, and Pearson correlation, was applied to further investigate the leakage relating to BLAKE-256, as this function is used within SPHINCS-256 several times with the same secret key but applied on different known addresses. Concerning fault attacks, using instances of one-time signature (OTS) or few-times signatures (FTS) to sign a same message has been shown to theoretically make many schemes, such as LD-OTS, W-OTS, and HORS, existentially forgeable with non-invasive attacks. These vulnerabilities are fatal for the Merkle signature schemes which implement the tree chaining method (CMSS). When the schemes provide $n/2 = 128$ bits of quantum security, a universal forgery can be created with around $q = 20$ different faulty signatures. This thesis demonstrates a practical application of fault attacks to create this universal forgery using voltage glitching on the previously mentioned ARM implementation of SPHINCS-256. An invasive attack performing key recovery against W-OTS by forcing bits of two quantities to be zero is also described. Countermeasures to thwart all the described attacks are discussed.

# Résumé

La présente thèse sonde l'état de l'art actuel de la cryptographie basée sur des fonctions de hachage en vue de repérer des vulnérabilités liées aux attaques par canaux cachés et attaques par fautes. Concernant les attaques par canaux cachés, nous avons analysé la consommation d'énergie d'un microcontrôleur Arduino Due sur lequel tourne une implémentation ARM personnalisée de SPHINCS-256 — le plus avancé des schémas de signatures digitales basés sur des fonctions de hachage. Nous avons employé de l'analyse simple de consommation sur une seule trace pour obtenir un premier aperçu de l'implémentation, puis sur plusieurs traces pour identifier une dépendance des données par rapport à la consommation d'énergie des fonctions impliquées dans SPHINCS-256. En se basant sur ces résultats, nous avons appliqué de l'analyse de consommation différentielle au moyen de différence de moyennes, tests de variance et corrélation de Pearson, afin d'investiguer la présence d'une fuite d'information de la fonction de hachage BLAKE-256 plus en profondeur, puisqu'elle est utilisée plusieurs fois dans le contexte de SPHINCS-256 avec une même clé secrète et appliquée à des adresses connues mais différentes. Concernant les attaques par fautes, le fait d'utiliser des instances de signatures à usage unique ou signatures à usage modéré pour signer plusieurs fois le même message rend, en théorie, beaucoup de schémas, comme LD-OTS, W-OTS et HORS, existentiellement falsifiables au moyen d'attaques non-envahissantes. Ces vulnérabilités sont fatales lorsqu'il est question du schéma de Merkle amélioré avec la méthode de chaînage d'arbres. Quand $n/2 = 128$ bits de sécurité quantique sont fournis, une falsification universelle peut être construite avec environ $q = 20$ signatures fautées différentes. La thèse démontre une application pratique d'attaques par fautes pour créer ladite falsification universelle en injectant un glitch dans la tension de l'appareil sur l'implémentation ARM de SPHINCS-256 susmentionnée. Une attaque envahissante permettant la récupération de la clé secrète du schéma W-OTS en annulant les bits de deux quantités à zéro est aussi décrite. Des contremesures pour parer toutes les attaques décrites sont discutées.

# Acknowledgments

I would like to express my gratitude to my thesis advisor Prof. Arjen Lenstra of the School of Computer and Communication Sciences at EPFL for the time he spent commenting my thesis, his fast e-mail responses, and all the trust he put in me through this semester. I would also like to thank my supervisor Dr. Hervé Pelletier at Kudelski Security for his continuous support in my project and the precious expertise he brought throughout the research.

Also, I must express my very profound gratitude to my colleagues at Kudelski Security for the friendly and welcoming environment, and especially to Dr. Roman Korkikian for all the help he provided to me when I was working on my side-channel analyses, and Andrew McLauchlan for helping me setting up the practical fault attack involved in this project. This thesis would never been what it is without their help.

Finally, I am immensely grateful to my parents, my girlfriend, and my friends, who never failed to support me during my whole studies. None of my accomplishments would have been possible without them, so with sincere gratitude, I would like to thank you all.

# Contents

# Chapter 1

# Introduction

## 1.1 Current cryptography

Nowadays, with the proliferation of computers and communication systems, information security has become an important concern. The increase of interconnected objects, such as personal laptops or smartphones, necessitates means to protect private information in digital form. The analysis of these means belongs to the field of *cryptography* which may be defined as the study of all the mathematical techniques related to the protection of information [MOV96].

The goal of cryptography is to adequately address information security notions, such as confidentiality, entity authentication, data integrity, and data origin authentication, in order to prevent cheating and other malicious activities. These notions are usually achieved using a number of cryptographic tools, called *primitives*, which provide the foundations for larger systems, called *cryptosystems*. Developing these primitives and assessing their security both in theory and practice is therefore the main task in cryptography, as they lie at the core of any digital protection [MOV96].

A common notion that cryptography is expected to provide is confidentiality. The primitive that corresponds to this feature is the *encryption scheme*, sometimes referred to as a *cipher*, which consists of turning clear messages into unintelligible ciphertexts according to an encryption key, and performing the opposite with a decryption key. In the context of a digital communication, encryption is often combined with message authentication, which is mainly achieved with a *digital signatures scheme*. Such a scheme produces authenticated quantities, called *digital signature*, that are bound to a message in such a way that the validity of the digital signature and its correspondence to the message can be verified.

Another primitive, quite specific to cryptography, is the *cryptographic hash function* which turns bit strings of arbitrary length into bit strings of some fixed length, called *hash values* or simply *digests*. Such functions are typically used for data integrity purposes, as hashed quantities serve as a compact and unpredictable identifier of an arbitrary string. They must however fulfill some qualitative properties; for instance, it must be difficult to go back from a hash value to its original input string, or to find two bit strings that produce the same digest. A whole family of digital signatures schemes solely based on these functions has been developped and will be studied in this thesis [MOV96].

### 1.1.1 Modern cryptosystems

Cryptography started way before the digital information era, as encryption methods were found in ancient Egypt 4,000 years ago [Kah96]. The most famous cryptographic method is probably Caesar's cipher, an encryption scheme which consists of shifting the letters from a text message over a certain number of positions in the alphabet. Nowadays, way more secure ciphers are used, such as the Advanced Encryption Standard (AES), an encrytion scheme that encrypts bit strings of fixed length.

The particularity of AES is that the encryption key is the same as the decryption key, resulting in a *secret-key encryption*. Therefore, in order to establish a confidential communication between two parties, the secret key needs to be shared. This problem was solved using *key exchange protocols* which rely on the intractability of hard problems, such as the Diffie–Hellman (DH) protocol which relies on the discrete logarithm problem.

This new security concept opened a whole new world of *public-key cryptography* that allows the deployment of cryptosystems using a public key known to everyone at all time, and a private key kept secret by the owner of the system. The most predominant public-key scheme currently used is the Rivest–Shamir–Adleman (RSA) public-key encryption which relies on the difficulty of factoring the product of two large prime numbers. Public-key cryptography also provided the necessary groundings for digital signature schemes. The RSA scheme is commonly used for this purpose, but the ElGamal signature scheme, a digital signature scheme based on the discrete logarithm as the DH key exchange, became popular with the use of elliptic curves, since it achieved the same level of security as RSA but with smaller keys.

Hash functions are most of the time found in digital signature schemes, as it allows the signing of fixed length message, but also in *commitment schemes*. In such schemes, a party commits to an arbitrary value by sending its hash to another party, and proves its commitment by revealing the initial value. The current standard is the Secure Hash Algorithm 2 (SHA-2) [Nat04], as an input collision was recently found in its predecessor SHA-1 [Ste+17]. However, using implementations of other hash functions is common, such as BLAKE [Aum+14] because of its notable speed.

The security of cryptosystems, such as RSA instances, is generally limited in time. This is mainly because of the constant elaboration of new attacks, forcing cryptographic schemes to be periodically updated. For instance, the development of a big enough quantum computer will make RSA and DH obsolete, as the next section will describe.

### 1.1.2 Post-quantum era

Modern computers are hardware devices which store bits and implement physical logic gates to perform arithmetic operations. Quantum computers, on their side, store quantum bits, known as *qubits*, and implement quantum gates to perform operations related to quantum mechanics. This new kind of technology opens a whole new computational world which is especially relevant for cryptography, as quantum computers can solve the hard problems on which the most popular public-key cryptosystems rely.

An example of a hard problem that quantum computers are able to solve is the factorization problem, using Shor's algorithm [Sho99]. Since the security of RSA is based on

the difficulty of this problem, solving it makes RSA insecure. Similarly, the discrete logarithm problem can also be solved with Shor's algorithm, making DH key exchange and ElGamal cryptosystems compromised as well. As in practice public-key cryptography is either based on the factorization or the discrete logarithm, quantum computers defeat every public-key cryptosystem currently established.

Secret-key cryptography is also affected by quantum computers, as Grover's algorithm, another quantum attack, is able to recover the input of a hash function or the secret key of a cipher in a time proportional to the square root of its size. To address this vulnerability, one needs to double the current bit size of secret keys and hash values, so the security is restored to its initial level.

It is not clear whether quantum computers storing large amount of stable qubits will be eventually developed, but public-key cryptosystems resistant to quantum computing need to be developed to address the risk. The study of these cryptosystems is called *post-quantum cryptography*, or sometimes *quantum-resilient cryptography*. Currently, it consists of five families that are independently researched: lattice-based cryptography, multivariate-based cryptography, code-based cryptography, isogeny-based cryptography, and hash-based cryptography which is the main topic of the current thesis.

Note that using quantum computers to develop cryptosystems, such as a quantum key exchange, is different from quantum-resilient cryptography. Post-quantum cryptosystems, as we understand them, are developed on *traditional* computers (i.e., non-quantum) to resist to the eventual arrival of quantum computers. The study of cryptosystems which exploit quantum mechanics principles offered by quantum computers is called *quantum cryptography* and is not addressed in this thesis.

## 1.2 Hardware attacks

Attacking cryptosystems to recover plaintexts or secret keys is not limited to theoretical or quantum attacks. Because processing data causes hardware to unintentionally leak information related to the data processed, physical devices running cryptographic procedures suffer from the leakage of secret inputs. This vector of attacks is called *hardware attacks* and is especially relevant when an adversary gains full physical access to the cryptographic device or its nearby environment.

Hardware attacks are classified in two main categories: passive attacks, and active attacks. The first category is basically based on observations, while the second involves manipulation to induce an exploitable abnormal behavior. Active attacks are moreover distinguished in three classes, depending on their level of tampering: non-invasive, semi-invasive, and invasive. Non-invasive attacks do not tamper with the device, semi-invasive attacks tamper with the device but leave no permanent damage, while invasive attacks induce a great level of tampering that may permanently damage the device. Invasive attacks are usually more effective, as they allow a better control of the circuit alterations.

### 1.2.1 Side-channel attacks

A Side-Channel Attack (SCA), in the context of hardware attacks, is a typical instance of passive attack in which an adversary monitors different physical variables to gain extra information on the secrets within a device. A straightforward hardware SCA, called Simple Power Analysis (SPA), consists of analyzing the power consumption of the device in order to discern patterns corresponding to specific operations. More advanced attacks, such as the Differential Power Analysis (DPA), combine the power consumption analysis with statistical strategies to increase their efficiency [KJJ99].

### 1.2.2 Fault attacks

A Fault Attack (FA) is an active type of hardware attack in which an adversary deliberately inserts an error, so the erroneous behavior of a cryptographic device can be exploited to recover secret information. Depending on the means and the skill of the attacker, the insertion can achieve different level of invasiveness. For instance, a non-invasive fault attack that requires a low cost and moderate skills is the introduction of glitches into the system clock signal to force instructions to be skipped. An example of a high-cost high-skill fault attack that stands at the limit of the semi-invasiveness is the laser beam injection, a focused laser beam that hits the logic gates of a circuit [BDL97].

## 1.3 Contribution

This thesis explores the current state of the art of digital signatures in hash-based cryptography in the context of hardware attacks. Even though these algorithms have been studied for their resistance against a quantum adversary, they will require to be implemented in existing technology. They are therefore exposed to the same physical attacks as for current cryptographic devices. In the context of deploying devices to survive the arrival of quantum computers, practical security needs to be assessed as much as possible, as there is a greater risk in attacking quantum-resilient cryptography using physical means than attacking modern cryptosystems with a large and fully operational quantum computer.

## 1.4 Structure

The thesis is structured as follows. Chapter 2 presents all the mathematical background required for the thesis. Chapter 3 makes a functional survey of hash-based algorithms up to the current state-of-the-art. Chapter 4 discusses the side-channel vulnerabilities of the described hash-based algorithms, while chapter 5 discusses of their vulnerabilities to fault injections. Chapter 6 concludes the thesis by summarizing the results found.

# Chapter 2

# Mathematical Background

## 2.1 Notation

Table 2.1 shows the different notations employed in the scope of this thesis.

| **Expression** | **Meaning** |
|---|---|
| $x \leftarrow y$ | Assignment of $x$ with $y$. |
| $x \sim \chi$ | The sample $x$ follows the distribution $\chi$. |
| $x \oplus y$ | Bit-wise XOR operations between $x$ and $y$. |
| $x \mid\mid y$ | Concatenation of $x$ with $y$. |
| $x \lll n$ | Left circular shift of $x$ by $n$ positions. |
| $\mathcal{U}(G)$ | Uniform random variable on set $G$. |
| $0^n$ or $1^n$ | Bit string of $n$ times 0 or 1. |
| $\{0,1\}^n$ | Set of all bit strings of length $n$. |
| $[x, y]$ | Set of integers from $x$ to $y$ ($x \leq y$). |
| $T[i]$ | Element $i$ of array $T$. |
| $x = (b_0, \ldots, b_{n-1})_2$ | Bit representation of $x$, where $b_0$ is the *most significant bit*. |
| $x[i{:}j] = (b_i, \ldots, b_{j-1})_2$ | Bit subset of $x$ ($0 \leq i < j \leq n$). |
| $\log x$ | Logarithm in base 2 of $x$. |
| $\texttt{Algorithm}(x)$ | Algorithm procedure with input $x$. |

**Table 2.1:** Notations.

## 2.2 Cryptographic notions

This section recalls broad cryptographic notions used through the entire thesis.

**Definition 2.1.** (Digital signature scheme) A *digital signature scheme* of security parameter $n \in \mathbb{N}$ is a cryptographic primitive which consists of three algorithms $\{\texttt{KeyGen}(1^n, r)$,

`Sign`($M$, $X$), `Verify`($M$, $\sigma$, $Y$)} that achieves *integrity* and *authenticity*. The algorithms are such that:

- `KeyGen`($1^n$, $r$) returns a *key pair* ($X$, $Y$) which consists of a private key $X$ and its corresponding public key $Y$. The private key $X$ was selected according to $r$ in a set of possible private keys.
- `Sign`($M$, $X$) produces the digital signature $\sigma$ corresponding to a message $M$ and a private key $X$.
- `Verify`($M$, $\sigma$, $Y$) checks the correctness of a produced signature $\sigma$ corresponding to a message $M$ according to a public key $Y$.

The *security parameter* $n$ defines the size in bits of the scheme and must be such that:

- `KeyGen`($1^n$, $r$), `Sign`($M$, $X$), `Verify`($M$, $\sigma$, $Y$) run in time polynomial in $n$.
- There is no probabilistic algorithm that can break the scheme in time polynomial in $n$.

When the private key has already been decided, the key generation algorithm `KeyGen` returns only the public key associated with it.

**Definition 2.2.** (Types of attack) [MOV96] The goal of an adversary is to *forge* signatures; that is, produce signatures which will be accepted as those of some other entity. The following provides a set of informal criteria for what it means to *break* a signature scheme.

- *Key recovery.* An adversary is either able to compute the private key information of the signer, or finds an efficient signing algorithm functionally equivalent to the valid signing algorithm.
- *Universal forgery.* An adversary is able to create a valid signature for a particular message or class of messages chosen a priori. Creating the signature does not directly involve the legitimate signer.
- *Existential forgery.* An adversary is able to forge a signature for at least one message. The adversary has little or no control over the message whose signature is obtained, and the legitimate signer may be involved in the deception.

## 2.3 Cryptographic functions

This sections summarizes the typical functions that are used through the thesis. Again, our definitions are informal.

**Definition 2.3.** (One-way function) [MOV96] A function $f$ from a set $A$ to a set $B$ is called a *One-Way Function* (OWF) if $f(x)$ is easy to compute $\forall\ x \in A$, but given a randomly chosen $y \in \mathrm{Im}(f)$ it is hard to find $x \in A$ such that $f(x) = y$.

**Definition 2.4.** (Compression function) [MOV96] A function $g$ from a set $A$ to a set $B$ is called a *Compression Function* (CF) when all the elements in $A$ are of fixed length $n$ and all the elements in $B$ are of fixed length $m$ and $n > m$.

**Definition 2.5.** (Random function) [MOV96] A function $f$ from a set $A$ to a set $B$ is called a *Random Function* (RF) if $\forall\ x \in A$, $f(x)$ is mapped to an element of $B$ in an "unpredictable" way.

The previous definitions can be combined.

**Definition 2.6.** (Hash function) [MOV96] A *hash function* is a function $h$ which fulfills at least the following two properties:

- *Compression*: For any $x$ of arbitrary finite bit length, $h$ maps an input $x$ to an output $h(x)$ of fixed bit length $m$.
- *Ease of computation*: Given $h$ and an input $x$, the computation of $h(x)$ is easy.

Specific secure applications often require the hash function to fulfill additional properties. The most basic properties are the following:

**Properties 2.7.** [MOV96]

1. *First-preimage resistance*: for essentially all pre-specified outputs, it is computationally infeasible to find any input which hashes to that output, i.e., to find any preimage $x$ such that $h(x) = y$ when given any $y$ for which a corresponding input is not known.
2. *Second-preimage resistance*: it is computationally infeasible to find any second input which has the same output as any specified input, i.e., given $x$, to find a 2nd-preimage $x' \neq x$ such that $h(x) = h(x')$.
3. *Collision resistance*: it is computationally infeasible to find any two distinct inputs $x$, $x'$ which hash to the same output, i.e., such that $h(x) = h(x')$. (Note that here there is free choice of both inputs.)

A hash function which fulfills these properties is known as a *cryptographic hash function*. *Breaking* a hash function means violating one of these three properties.

**Definition 2.8.** (Pseudorandom number generator) [MOV96] A *Pseudo-Random Number Generator* (PRNG) is a deterministic algorithm $G_\lambda$ which, given a random bit string of length $n$, referred to as the *seed*, outputs a binary sequence of length $\lambda(n)$ which is "indistinguishable" from a uniform distribution.

**Definition 2.9.** (Pseudorandom function ensemble) [Gol01] A *Pseudo-Random Function ensemble* (PRF) is a sequence $F_\lambda = \{F_n\}_{n \in N}$ of random variables such that $F_n$ assumes values in the set of functions that, given a bit strings of length $\lambda(n)$, output a bit strings of length $n$ which is "indistinguishable" from a uniform distribution.

# Chapter 3

# Hash-based Cryptography

The hash-based cryptography family consists of cryptographic primitives whose resistance is mostly based on the security properties of a hash function. This class of algorithms can only provide authentication schemes, since encryption requires to turn a ciphertext back into a plaintext which undermines the very properties of hash functions. However, because of their robustness against quantum computing, this category turns out to be an interesting alternative for a replacement of digital signatures based on non-quantum resistant problems.

Generally speaking, the signatures in this family are performed in a commitment fashion: the signer draws a set of secret values at random, hashes them at least once, and publishes the hashed values as a public key which is assumed to be authenticated and available at all time. Signing a message consists of revealing a specific set of secret values while keeping others private. A verifier checks the correctness of the signature by hashing the secret values revealed and by comparing the result with the corresponding values from the public key.

This signing method is sometimes combined with the construction of a binary hash tree, which allows the signer to sign multiple messages with a single public key. In this case, the signature not only consists of the secret values, but also of the nodes of the tree needed to construct its root. To verify that all the messages come from the same signer, a verifier computes the root of the hash tree using the corresponding signatures and compares it with the public value.

There are many reasons why to choose hash-based cryptography for digital signatures. First of all, unlike modern cryptosystems, these algorithms do not rely on a hard problem, but only on the security of the underlying hash function. As a result, they can replace most of current applications, since cryptographic hash functions are already a cornerstone of any digital signature scheme. Also, the security of most of the schemes has been proved to be reduced to the properties of the used hash function, meaning that breaking the schemes amounts to breaking the hash function. Since the schemes work with any hash function, an obsolete hash function can therefore be replaced with a more recent one to keep their security up to date. This field being well studied, it makes the hash-based schemes convincingly secure from an objective point of view, which is why this cryptographic family is the most promising approach for digital signature replacement.

While hash-based cryptography looks favorable for a digital signatures replacement, the computation time and memory requirements of most of the schemes often limit their practical use. Besides, the designs for these are mostly stateful and limited in the number of signatures. Moreover, the practicality of the main schemes does not scale as much as RSA or AES, since they mainly work "per bit". The signature and public key sizes can also be perceived as a negative aspect of the scheme, as lighter schemes exist. However, many improvements have been made to mitigate these disavantages, leading to acceptable solutions.

Hash-based cryptography has been first investigated in 1979 by Leslie Lamport with the famous Lamport–Diffie one-time signature scheme [Lam79]. A bit later, Ralph Merkle developed the Merkle signature scheme [Mer89] which transforms a one-time signature scheme into a stateful multiple-time signature scheme. He also introduced the Winternitz one-time signature variant [Mer89] which generalizes the concept from the Lamport–Diffie scheme. The most important security result was obtained by John Rompel who showed in 1990 that one-way functions are necessary and sufficient for secure hash-based signatures [Rom90]. A third type of hash-based signatures emerged in 2002, when the father and son Leonid and Natan Reyzin invented HORS [RR02], a scheme whose security progressively degrades with the number of signatures made.

Merkle's authentication scheme being inadequate due to its stateful nature, Oded Goldreich conceived in 2004 a similar scheme [Gol04] in which the message digest decides the leaf to be addressed in the tree—an idea that he originally applied to a totally different algorithm [Gol86]. Even with this improvement, hash-based schemes were still very costly and inapplicable to real case scenarios, so many other designs came up to address the issue. For instance, Johannes Buchmann et al. created CMSS [Buc+06] in 2006, a technique that consists of chaining Merkle's authentication schemes together. Two years after, Erik Dahmen et al. found a way to make Merkle's signature scheme resistant to second preimage attacks rather than collision attacks [Dah+08]. All these enhancements were combined in 2014 when Daniel J. Bernstein et al. developed SPHINCS [Ber+14], a stateless high-security digital signatures scheme based on hash functions. Finally, let us mention the two signatures schemes made out of non-interactive zero-knowledge proofs, called Fish, and Picnic [Cha+17], which are not going to be studied in this thesis.

This chapter starts by introducing the three basic ways of creating a hash-based digital signature scheme: namely, the one-time signatures, the multiple times signatures, and the few-times signatures. It then concludes with a presentation of the SPHINCS cryptosystem, which covers all the improvements required to create what is considered as the current state of the art.

## 3.1 One-time signatures

Hash-based One-Time Signature schemes (OTS) describe ways of signing only one message with one public key, using only a hash function as a cryptographic primitive. The security of such schemes is mainly based on the collision resistance of the hash function used and on the fact that if the same key pair is used to sign more than one unique message, an attacker can forge a valid signature for another message. The idea is to publish the hash of multiple secret values as a public key and to reveal a few of them to sign a message,

leading thus to a trivial precursor of true public key cryptosystems where the same secret key can never be used forever and has to be released while still allowing effective signature verification.

### 3.1.1 Lamport–Diffie one-time signature scheme

The Lamport–Diffie One-Time Signature scheme (LD-OTS) is the first one-time signature scheme to have been established. The scheme involves hashing two secret values per bit in the message digest, so the signature consists of revealing the one corresponding to the value of each bit from the actual message digest. A small example illustrates the process in Figure 3.1 where a digest of $n = 4$ bits is signed.

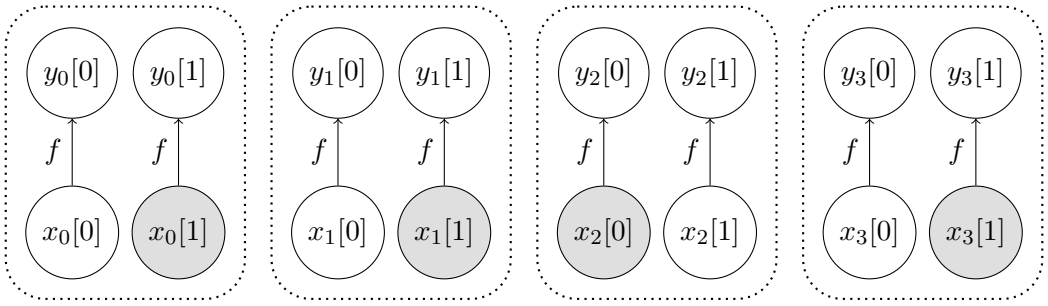



**Figure 3.1:** Example of a Lamport–Diffie one-time signature scheme. The signature consists of the highlighted nodes.

**Setup** An LD-OTS instance of security parameter $n \in \mathbb{N}$ requires the following public parameters:

- $f : \{0,1\}^n \to \{0,1\}^n$ : a random one-way function
- $h : \{0,1\}^* \to \{0,1\}^n$ : a cryptographic hash function.

With these parameters, the scheme achieves $n/2$ bits of quantum security, limited to the collision-resistance of the random one-way function.

**Key Generation** In order to sign every possibility for an $n$-bit digest, $2n$ secret values need to be randomly drawn. The signer then commits to these values by hashing them with the random one-way function and publishing the result as the public key of the scheme.

The private key $X$ is generated as follows:

$$X \leftarrow \begin{pmatrix} x_0[0] & x_0[1] \\ x_1[0] & x_1[1] \\ \vdots & \vdots \\ x_{n-1}[0] & x_{n-1}[1] \end{pmatrix}$$

where $x_i[b] \sim \mathcal{U}(\{0,1\}^n)$ for $0 \leq i < n$ and $b \in \{0,1\}$.

The public key $Y$ is generated as follows:

$$Y \leftarrow \begin{pmatrix} y_0[0] & y_0[1] \\ y_1[0] & y_1[1] \\ \vdots & \vdots \\ y_{n-1}[0] & y_{n-1}[1] \end{pmatrix}$$

where $y_i[b] = f(x_i[b])$ for $0 \leq i < n$ and $b \in \{0, 1\}$.

**Signing Procedure** To sign a message, the signer proceeds by revealing the secret values which correspond to the value of each bit from the digest of the message.

The LD-OTS signing procedure for a message $M \in \{0,1\}^*$ under the key pair $(X, Y)$ is described in Algorithm 3.1.1.

**input :** $M \in \{0,1\}^*$ – the message
**input :** $X = (x_i[0], x_i[1])$ – the private key
**output :** $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_{t-1})$ – the LD-OTS signature associated with $M$

---

1 Compute the digest $d \leftarrow h(M)$
2 Let $(d_0, d_1, \ldots, d_{n-1})_2$ be the binary representation of $d$
3 **for** $i = 0$ **to** $n - 1$ **do**
4 $\quad \sigma_i \leftarrow x_i[d_i]$
5 **end**
6 **return** $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_{n-1})$

**Algorithm 3.1.1:** LD-OTS signing procedure.

**Verification Procedure** To verify the correctness of a signature, a verifier opens the commitments with the received secret values from the signature. If the position of the resulting hash inside the public key corresponds to the binary representation of the message digest, then it means that the signature is correct.

The LD-OTS verification procedure for a signature $\sigma \in \{0,1\}^{(n \times n)}$ associated with the message $M \in \{0,1\}^*$ under the key pair $(X, Y)$ is described in Algorithm 3.1.2.

### 3.1.2 Winternitz one-time signature scheme

The Winternitz One-Time Signature scheme (W-OTS) is a natural generalization of the LD-OTS which drastically improves the size of the signatures. Instead of choosing one secret value per bit—as it was done in LD-OTS—the idea is to hash values in chain starting from one secret value. As a result, several bits can be signed by revealing only one value from a specific point in the chain.

In addition to the signature of each bit strings from the message digest, the scheme also signs a certain checksum. This prevents an attacker from exploiting all the points in the chain between the ones revealed and the public key. This checksum is such that an

attacker has enough knowledge to sign either other message digests or other checksums, but never both at the same time.

The W-OTS signing process is shown in a small example in Figure 3.2 where a digest of $n = 6$ bits is split in blocks of $w = 2$ bits. The chains on the left correspond to the message blocks, while the chains on the right correspond to the checksum blocks.

**input** : $M \in \{0,1\}^*$ – the message
**input** : $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_{n-1})$ – the LD-OTS signature associated with $M$
**input** : $Y = (y_i[0], y_i[1])$ – the public key
**output** : `True` if the signature is correct, `False` otherwise

---

Compute the digest $d \leftarrow h(M)$
Let $(d_0, d_1, \ldots, d_{n-1})_2$ be the binary representation of $d$
**for** $i = 0$ **to** $n - 1$ **do**
    **if** $f(\sigma_i) \neq y_i[d_i]$ **then**
        **return** `False`
    **end**
**end**
**return** `True`

**Algorithm 3.1.2:** LD-OTS verification procedure.

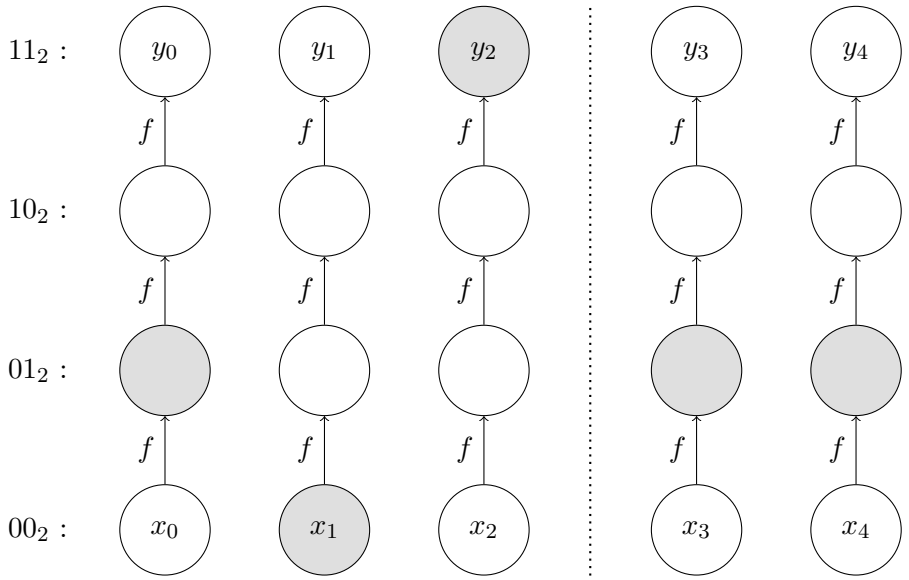
**Setup** A W-OTS instance of security parameter $n \in \mathbb{N}$ requires the following public parameters:

- $w \in \mathbb{N}$ : the window of bits to be signed simultaneously
- $f : \{0,1\}^n \to \{0,1\}^n$ : a random one-way function
- $h : \{0,1\}^* \to \{0,1\}^n$ : a cryptographic hash function.

The parameter $w$ allows a tradeoff between signature size and computation time. Indeed, since the signature size decreases linearly in $w$, big values for $w$ produce short signatures. However, key generation, signing, and verification procedures are exponential in $w$, so small values for $w$ lead to faster computations. The optimal value depends therefore on the resources available to the signer and the verifier.

Moreover, the scheme requires the definition of the following quantities:

- $W = 2^w$ : the length of the chain
- $\ell_1 = \left\lceil \frac{n}{w} \right\rceil$ : the number of bit strings in a digest
- $\ell_2 = \left\lceil \frac{\lfloor \log \ell_1 \rfloor + 1 + w}{w} \right\rceil$ : the number of bit strings in the checksum
- $\ell = \ell_1 + \ell_2$ : the total number of bit strings to sign.




$$M = 01\,00\,11_2 \qquad C = \sum_{i=0}^{2}(11_2 - b_i) = 01\,01_2$$

**Figure 3.2:** Example of a Winternitz one-time signature scheme. The signature consists of the highlighted nodes.

**Key Generation** In order to sign each possibility of $w$-bit strings in the message digest and the checksum, the signer uniformly draws $\ell$ secret values to establish the starting point of the hash chains.

The private key $X$ is therefore generated as follows:

$$X \leftarrow \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{\ell-1} \end{pmatrix}$$

where $x_i \sim \mathcal{U}(\{0,1\}^n)$ for $0 \leq i < \ell$.

The chaining method consists of repeatedly applying the one-way function to the result of the previous application, starting from the secretly drawn values. Only the final point of the chain is published, while every other point in the chain is kept secret.

The public key $Y$ is therefore generated as follows:

$$Y \leftarrow \begin{pmatrix} c^{W-1}(x_0) \\ c^{W-1}(x_1) \\ \vdots \\ c^{W-1}(x_{\ell-1}) \end{pmatrix}$$

where $c^i(x)$ represents the chaining function which is defined as follows:

$$c^i(x) = \begin{cases} x & \text{if } i = 0 \\ f(c^{i-1}(x)) & \text{if } i > 0. \end{cases}$$

**Signing Procedure** In order to sign a message, the signer first computes the digest of the message. The actual data to be signed consists of the digest extended by a particular checksum.

Once the message digest and the checksum have been computed, they are written as the concatenation of length-$w$ bit strings. The signer reveals the points in the chain whose position correspond to the value of each bit string.

The W-OTS signing procedure for a message $M \in \{0,1\}^*$ under the key pair $(X, Y)$ with a window size of $w$ is described in Algorithm 3.1.3.

**input :** $M \in \{0,1\}^*$ – the message
**input :** $X = (x_0,\ x_1,\ \ldots,\ x_{\ell-1})$ – the private key
**output :** $\sigma\ = (\sigma_0,\ \sigma_1,\ \ldots,\ \sigma_{\ell-1})$ – the W-OTS signature associated with $M$

---

1 Compute the digest $d \leftarrow h(M)$
2 Let $(b_0,\ b_1,\ \ldots,\ b_{\ell_1-1})$ be the result of splitting $d$ in blocks of $w$ bits
3 Compute the checksum $C \leftarrow \sum_{i=0}^{\ell_1-1} (W - b_i)$
4 Let $(b_{\ell_1},\ b_{\ell_1+1},\ \ldots,\ b_{\ell_1+\ell_2})$ be the result of splitting $C$ in blocks of $w$ bits
5 **for** $i = 0$ **to** $\ell - 1$ **do**
6 $\quad \sigma_i \leftarrow c^{b_i}(x_i)$
7 **end**
8 **return** $\sigma = (\sigma_0,\ \sigma_1,\ \ldots,\ \sigma_{\ell-1})$

**Algorithm 3.1.3:** W-OTS signing procedure.

**Verification Procedure** To verify the correctness of a signature, a verifier proceeds as the signer and computes the message digest along with the checksum. If applying the one-way function on the received values as many times as expected returns the entire public key, then it means that the signature is correct.

The W-OTS verification procedure for a signature $\sigma \in \{0,1\}^{(\ell \times n)}$ associated with the message $M \in \{0,1\}^*$ with a window size of $w$ under the key pair $(X, Y)$ is described in Algorithm 3.1.4.

## 3.2 Multiple times signatures

The one-time signature schemes present a secure way of signing a message, but the fact that only one message can be signed with one key pair is not suitable for real-life scenarios. Ralph Merkle solved the issue by designing the Merkle's signature scheme [Mer89] which

**input** : $M \in \{0,1\}^*$ – the message
**input** : $\sigma = (\sigma_0,\ \sigma_1,\ \ldots,\ \sigma_{\ell-1})$ – the W-OTS signature associated with $M$
**input** : $Y = (y_0,\ y_1,\ \ldots,\ y_{\ell-1})$ – the public key
**output** : `True` if the signature is correct, `False` otherwise

---

Compute the digest $d \leftarrow h(M)$
Let $(b_0,\ b_1,\ \ldots,\ b_{\ell_1-1})$ be the result of splitting $d$ in blocks of $w$ bits
Compute the checksum $C \leftarrow \sum_{i=0}^{\ell_1-1}(W - b_i)$
Let $(b_{\ell_1},\ b_{\ell_1+1},\ \ldots,\ b_{\ell_1+\ell_2})$ be the result of splitting $C$ in blocks of $w$ bits
**for** $i = 0$ **to** $\ell - 1$ **do**
    **if** $c^{W-b_i-1}(\sigma_i) \neq y_i$ **then**
        **return** `False`
    **end**
**end**
**return** `True`

**Algorithm 3.1.4:** W-OTS verification procedure.

allows a signer to sign multiple messages with a single key pair. The scheme makes use of the construction of a complete binary hash tree whose root is the only public value.

The signing process is usually done in two phases: first, the signer authenticates the message with an instance of a one-time signature scheme. Second, the signer authenticates the verification key from the instance with the computation of the root of the binary hash tree. By doing this, the validity of a key pair is extended, but is still limited in the number of instances available, since the tree is finite.

### 3.2.1 Merkle's tree authentication scheme

The original hash-based multiple times signature scheme is the Merkle's Signature Scheme (MSS). The scheme considers a binary tree whose leaves are instances of a one-time signature scheme. Each node of the tree is constructed with the hash of its two children put together, up until the root of the tree which is published as the public key. The signer signs the messages using a one-time signature instance which will be authenticated using the nodes necessary to construct the root. The full signature therefore consists of the one-time signature of the instance used, along with the sequence of authentication nodes from the tree. Note that since the signer needs to avoid to use the instances of one-time signature multiple times, an indicator (the state) needs to be kept track of.

**Definition 3.1.** (Merkle tree) A *Merkle tree* of height $H \in \mathbb{N}$ is a binary tree $\nu$ whose nodes $\nu_i$ at altitude[1] $1 \leq i \leq H$ and index $0 \leq j \leq 2^{H-i}$ are defined as follows:

$$\nu_i[j] = g(\nu_{i-1}[2j] \,||\, \nu_{i-1}[2j+1])$$

where $g : \{0,1\}^{2n} \to \{0,1\}^n$ is a random one-way compression function of security parameter $n \in \mathbb{N}$.

An illustration of a Merkle tree authentication scheme is shown on Figure 3.3. In this example, the Merkle tree used is of height $H = 3$, providing up to $2^H = 8$ instances of one-time signatures. The root of the tree $\nu_3[0]$ has been computed once and is assumed to be public.

Now, suppose that a message $M$ is being signed with the fourth instance of the one-time signature scheme whose key pair is $(X_3, Y_3)$. This provides the signature $(Y_3, \sigma_3)$, but to complete the signature, the signer needs to authenticate $Y_3$ with the tree. Since the fourth leaf $\nu_0[3]$ actually consists of the hash of $Y_3$, a verifier is able to re-construct the Merkle tree root as long as the highlighted nodes ($\nu_0[2]$, $\nu_1[0]$, $\nu_2[1]$) are known. These nodes are referred to as the *authentication path* of the leaf. Therefore, the final signature will also include them, so the root can be computed and compared to the public key.

We clearly see in the above example that the scheme has two main downsides. The first one is that after $2^H$ messages, the signer cannot sign any additional message, since no more OTS instance is available. Besides, the tree cannot be extended without changing the public-key. The second one is that the signer needs to remember what instances of OTS have already been used. This characterizes the scheme as *stateful*; the state being the leaf to be used next.

**Setup** An MSS instance of security parameter $n \in \mathbb{N}$ requires the following public parameters:

- $H \in \mathbb{N}$ : the height of the Merkle tree
- $g : \{0,1\}^{2n} \to \{0,1\}^n$ : a random one-way compression function
- $h : \{0,1\}^* \to \{0,1\}^n$ : a cryptographic hash function
- $(\mathtt{KeyGen}_{\mathtt{OTS}}, \mathtt{Sign}_{\mathtt{OTS}}, \mathtt{Verif}_{\mathtt{OTS}})$ : any one-time signature scheme.

**Key Generation** In order to sign $2^H$ messages, the signer first needs to construct the Merkle tree. To achieve this, all the instances of the one-time signatures need to be instantiated. Assume the one-time signature scheme requires $m = \text{poly}(n)$ random values:

$$\begin{array}{rcl@{\qquad}rcl} X_0 & \sim & \mathcal{U}(\{0,1\}^{m\times n}) & Y_0 & \leftarrow & \mathtt{KeyGen}_{\mathtt{OTS}}(1^n, X_0) \\ X_1 & \sim & \mathcal{U}(\{0,1\}^{m\times n}) & Y_1 & \leftarrow & \mathtt{KeyGen}_{\mathtt{OTS}}(1^n, X_1) \\ \vdots & & & \vdots & & \\ X_{2^H-1} & \sim & \mathcal{U}(\{0,1\}^{m\times n}) & Y_{2^H-1} & \leftarrow & \mathtt{KeyGen}_{\mathtt{OTS}}(1^n, X_{2^H-1}). \end{array}$$

[1]The height of the node in the tree is referred to as the altitude to prevent the confusion with other meanings.

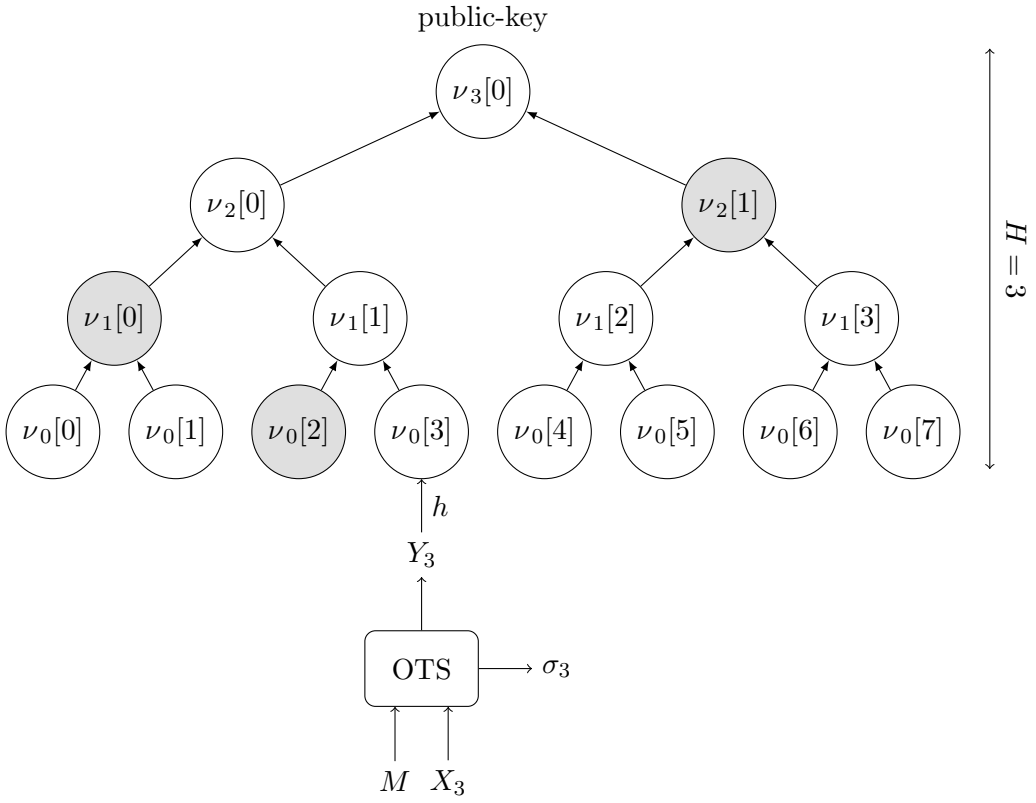



**Figure 3.3:** Example of a Merkle's tree authentication scheme. In addition to the one-time signature $\sigma_3$, the highlighted nodes are included in the signatures.

The private key $X$ consists of the one-time signature secret keys:

$$X \leftarrow \begin{pmatrix} X_0 \\ X_1 \\ \vdots \\ X_{2^H-1} \end{pmatrix}.$$

Once the OTS keys have been produced, the Merkle tree needs to be generated from the one-time signature verification keys. This is typically computed as in Algorithm 3.2.1.

The public key $Y$ consists of the root of the Merkle tree:

$$Y \leftarrow \nu_H[0].$$

In addition to the keys generation, the state of the scheme also needs to be set up. This can simply be achieved by initializing a counter $s \leftarrow 0$ which will be increased each time a message has been signed.

**Signing Procedure** In order to sign the $s^{\text{th}}$ message $M_s$ (where $0 \leq s < 2^H$), the signer will use the $s^{\text{th}}$ instance of the one-time signatures $(X_s, Y_s)$ to produce the one-time signature $(Y_s, \sigma_s)$. This signature will be extended with the authentication path

**input :** $(Y_0, Y_1, \ldots, Y_{2^H-1})$ – the OTS verification keys
**output :** $\nu$ – the Merkle tree nodes

---

Let the leaves $(\nu_0[0], \nu_0[1], \ldots, \nu_0[2^H - 1])$ be $(h(Y_0), h(Y_1), \ldots, h(Y_{2^H-1}))$
**for** $i = 1$ **to** $H$ **do**
  **for** $j = 0$ **to** $2^{H-i} - 1$ **do**
    $\nu_i[j] \leftarrow g(\nu_{i-1}[2j] \,||\, \nu_{i-1}[2j+1])$
  **end**
**end**
**return** $\nu$

**Algorithm 3.2.1:** Merkle tree construction algorithm.

$(a_0, a_1, \ldots, a_{H-1})$. This sequence needs to regroup the siblings of each computed node starting from the $s^{\text{th}}$ leaf.

The MSS signing procedure for a message $M_s \in \{0,1\}^*$ under the key pair $(X, Y)$ is described in Algorithm 3.2.2.

**input :** $M_s \in \{0,1\}^*$ – the message
**input :** $X = (X_0, X_1, \ldots, X_{2^H-1})$ – the private key
**input :** $\nu$ – the Merkle tree nodes
**intern :** $s$ – state of the scheme (counter)
**output :** $\sigma_s = (s, \sigma_{\text{OTS}}, Y_{\text{OTS}}, (a_0, a_1, \ldots, a_{H-2}))$ – the MSS signature

---

$(\sigma_{\text{OTS}}, Y_{\text{OTS}}) \leftarrow \texttt{Sign}_{\texttt{OTS}}(M_s, X_s)$
**for** $i = 0$ **to** $H - 2$ **do**
  $a_i \leftarrow \begin{cases} \nu_i[s/2^i - 1] & \text{if } \lfloor s/2^i \rfloor \equiv 1 \pmod 2 \\ \nu_i[s/2^i + 1] & \text{if } \lfloor s/2^i \rfloor \equiv 0 \pmod 2 \end{cases}$
**end**
$s \leftarrow s + 1$ `/* Increase internal counter */`
**return** $\sigma_s = (s, \sigma_{OTS}, Y_{OTS}, (a_0, a_1, \ldots, a_{H-2}))$

**Algorithm 3.2.2:** MSS signing procedure.

**Verification Procedure** To verify the correctness of a signature, a verifier first checks the validity of the one-time signature received. Then, the Merkle tree root is computed using the hash of the verification key at the given leaf index and the authentication path. If the result corresponds to the public key, then the signature is considered as correct.

The MSS verification procedure for a signature $\sigma_s = (s, \sigma_{\text{OTS}}, Y_{\text{OTS}}, (a_0, a_1, \ldots, a_{H-1}))$ associated with the message $M \in \{0,1\}^*$ under the key pair $(X, Y)$ is described in Algorithm 3.2.3.

**input** : $M_s \in \{0,1\}^*$ – the message
**input** : $\sigma_s = (s, \sigma_{\text{OTS}}, Y_{\text{OTS}}, (a_0, a_1, \ldots, a_{H-1}))$ – the MSS signature
**input** : $Y$ – the public key
**output** : `True` if the signature is correct, `False` otherwise

---

**if** $\texttt{Verif}_{\texttt{OTS}}(M_s, Y_{\text{OTS}}, \sigma_{\text{OTS}})$ **is** `False` **then**
  **return** `False`
**end**
$R \leftarrow h(Y_{\text{OTS}})$
**for** $i = 0$ **to** $H - 1$ **do**
  $R \leftarrow \begin{cases} g(a_i \mid\mid R) & \text{if } \lfloor s/2^i \rfloor \equiv 1 \pmod 2 \\ g(R \mid\mid a_i) & \text{if } \lfloor s/2^i \rfloor \equiv 0 \pmod 2 \end{cases}$
**end**
**return** $R \stackrel{?}{=} Y$

**Algorithm 3.2.3:** MSS verification procedure.

## 3.3 Few-times signatures

A Few-Times Signatures scheme (FTS) allows a signer to sign a few messages with the same public key such that the security of the scheme degrades with the amount of messages signed. The idea is very similar to the one-time signature schemes: commit to a subset of secret values with a one-way function and reveal a few of them corresponding to the message digest. The difference comes from the fact that the subset is slightly bigger, making the selection of secret values that must be revealed more flexible depending on the number of signatures. The security however weakens with the number secret values revealed.

### 3.3.1 HORS

The HORS scheme [RR02] stands for "Hash to Obtain Random Subsets" and was originally intended to be a better version of the BiBa scheme [Per01] (which stands for "Bins and Balls"). The idea behind HORS is to sign every possibility of fixed-length blocks of bits, rather than signing the bits of the message digest. In this case, the public key consists of the hash of a secret value for each possible block, while a signature consists of partitioning the digest to sign in multiple blocks and revealing the secret values corresponding to them.

It is easy to see that since the same subset is being used, the security of the scheme weakens with the number of signed messages. However, if the blocks are long enough, the selection is said to be *q-subset-resilient* if any kind of attacker is still unable to find a message such that its digest is a combination of the known blocks, when $q$ signatures are known. For a non-adaptive chosen-message attacker, the corresponding bits of security of an instance is $k(\tau - \log k - \log q)$, where $q$ is the number of reuse, $k$ the number of blocks to sign, and $\tau$ the blocks bit length. This expression is derived from the probability that

after $qk$ elements are fixed, $k$ elements chosen at random form a subset of them; an upper bound on the actual chance of success [RR02].

**Setup** A HORS instance of security parameter $n \in \mathbb{N}$ requires the following public parameters:

- $f : \{0,1\}^n \to \{0,1\}^n$ : a random one-way function
- $h : \{0,1\}^* \to \{0,1\}^n$ : a cryptographic hash function
- $k$ : the number of blocks to sign.

Moreover, let $\tau = \lceil n/k \rceil$ be the number of bits inside one block, similar to the parameter $w$ in W-OTS.

**Key Generation** In order to sign every possibility for a $\tau$-bit block, the signer needs to randomly pick $t = 2^\tau$ secret values. These values will then be hashed with the one-way function and the result will be made public.

The private key $X$ is therefore generated as follows:

$$X \leftarrow \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{t-1} \end{pmatrix}$$

where $x_i \sim \mathcal{U}(\{0,1\}^n)$ for $0 \leq i < t$.

The public key $Y$ is therefore generated as follows:

$$Y \leftarrow \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{t-1} \end{pmatrix}$$

where $y_i = f(x_i)$ for $0 \leq i < t$.

**Signing Procedure** To sign a message, the signer splits the message digest in $k$ blocks, and reveals the secret values corresponding to the value of each of these blocks.

The HORS signing procedure for a message $M \in \{0,1\}^*$ under the key pair $(X, Y)$ with the number of blocks $k$ is described in Algorithm 3.3.1.

**Verification Procedure** To verify the correctness of a signature, a verifier splits the message digest in $k$ blocks and compares the hash of the values revealed in the signature with the ones in the public key that correspond to the digest blocks.

The HORS verification procedure for a signature $\sigma \in \{0,1\}^{(k \times n)}$ associated with the message $M \in \{0,1\}^*$ under the key pair $(X, Y)$ with the amount of blocks $k$ is described in Algorithm 3.3.2.

**input** : $M \in \{0,1\}^*$ – the message
**input** : $X = (x_0,\ x_1,\ \ldots,\ x_{t-1})$ – the private key
**output** : $\sigma\ = (\sigma_0,\ \ldots,\ \sigma_{k-1})$ – the HORS signature associated with $M$

---

Compute the digest $d \leftarrow h(M)$
Let $(b_0,\ \ldots,\ b_{k-1})$ be the result of partitioning $d$ in blocks of $\tau$ bits
**for** $i = 0$ **to** $k - 1$ **do**
    $\sigma_i \leftarrow x_{b_i}$
**end**
**return** $\sigma = (\sigma_0,\ \ldots,\ \sigma_{k-1})$

**Algorithm 3.3.1:** HORS signing procedure.

**input** : $M \in \{0,1\}^*$ – the message
**input** : $\sigma\ = (\sigma_0,\ \ldots,\ \sigma_{k-1})$ – the HORS signature associated with $M$
**input** : $Y = (y_0,\ y_1,\ \ldots,\ y_{t-1})$ – the public key
**output** : `True` if the signature is correct, `False` otherwise

---

Compute the digest $d \leftarrow h(M)$
Let $(b_0,\ \ldots,\ b_{k-1})$ be the result of partitioning $d$ in blocks of $\tau$ bits
**for** $i = 0$ **to** $k - 1$ **do**
    **if** $f(\sigma_i) \neq y_{b_i}$ **then**
        **return** `False`
    **end**
**end**
**return** `True`

**Algorithm 3.3.2:** HORS verification procedure.

## 3.4 SPHINCS

When it comes to a drop-in replacement for stateless digital signatures using hash-based schemes, SPHINCS ("Stateless Practical Hash-based Incredibly Nice Collision-resilient Signatures") is probably the cryptosystem which represents the top of the current state of the art. It was designed in 2014 by Daniel J. Bernstein et al. [Ber+14] to show that stateless hash-based digital signature are practical, even for a high standard of security and performance. The scheme uses an enhanced model of a Merkle tree made of one-time and few-times signatures, so a virtually unlimited set of messages can be securely signed.

The SPHINCS digital signatures scheme regroups many improvements that have been made over the past years. First, it drastically reduces the secret key size by using a pseudorandom number generator to recover the one-time signature secret keys when needed, rather than having them stored all the time. Then, it extends the original Merkle sig-

nature design using hypertrees—an abstraction which drastically improves the key generation and signature times. Also, it removes the state of the algorithm by addressing the one-time signature instances to sign the messages as a function of the message digest. Furthermore, the security of the scheme is reduced to second preimage attacks rather than collision attacks by annexing a layer of XOR operations on top of the input of every hash function. The scheme replaces the instances of the one-time signatures intended for the messages with instances of a better few-times signatures scheme, so even if the full tree is known, there is still a subset-resilience to fight. Finally, the values from the one-time signature verification keys are compressed using L-Trees—an unbalanced variant of Merkle trees.

The section will cover all the aforementionned improvements before describing the concrete construction of the SPHINCS cryptosystem.

### 3.4.1 Using a pseudorandom number generator

The Merkle authentication scheme requires an amount of one-time signature secret keys exponential in the height of the Merkle tree. This is unpractical for current security standards, since most of applications cannot store such an amount of data at once. A solution, initially developed by Luis C. Coronado García in 2005 as a first improvement of Merkle's scheme in his thesis [Gar05], consists of using a pseudorandom number generator to produce the random data needed for the secret keys on the fly. In this case, the signer stores only a seed which, when fed to the PRNG, produces a stream of data whose first values represent the OTS secret keys.

This improvement actually affects both Merkle's authentication scheme and the underlying one-time signature scheme. Indeed, the secret values in the OTS secret key are in fact replaced by a single seed which will be used to recover them. However, since MSS uses many instances of OTS, each of their seeds will also be the result of a pseudorandom generation using another seed that will just be updated after each use.

In order to put this improvement into practice, the Merkle authentication scheme and the one-time signature scheme need to include the following:

- $\text{PRNG} : \{0,1\}^n \rightarrow \{0,1\}^n \times \{0,1\}^n$ : a pseudorandom number generator.

The one-time signature key generation now consists of producing (or using) a seed $\text{SEED}_{\text{OTS}}$ and calling the pseudorandom generator with it to draw the secret values required for the secret keys. This allows the generation of secret keys without the need of storing the secret values altogether.

The Merkle authentication scheme, for its part, also involves generating one general seed $\text{SEED}_{\text{MSS}}$ that will be used to create the instances of OTS. Thanks to this, the overall secret key for this improved version of MSS consists only of one random value:

$$X \leftarrow \text{SEED}_{\text{MSS}}$$

where $\text{SEED}_{\text{MSS}} \sim \mathcal{U}(\{0,1\}^n)$.

In order to compute the root of the Merkle tree, so that the overall public key can be published, the signer needs to calculate the leaves of the Merkle tree. As in the

original scheme, these leaves consist of the hash of the OTS verification keys, and the tree is constructed as in Definition 3.1. To recover the OTS verification keys, the PRNG will be repeatedly called using the overall secret seed to the different $\text{SEED}_{\text{OTS}_i}$ for $0 \leq i < 2^H$. These resulting seeds are then independently used to draw all the necessary OTS secret values (assume there are $m$ of them), so that OTS verification keys can be computed (following the procedure corresponding to the one-time signature scheme chosen). Figure 3.4 illustrates the above process.

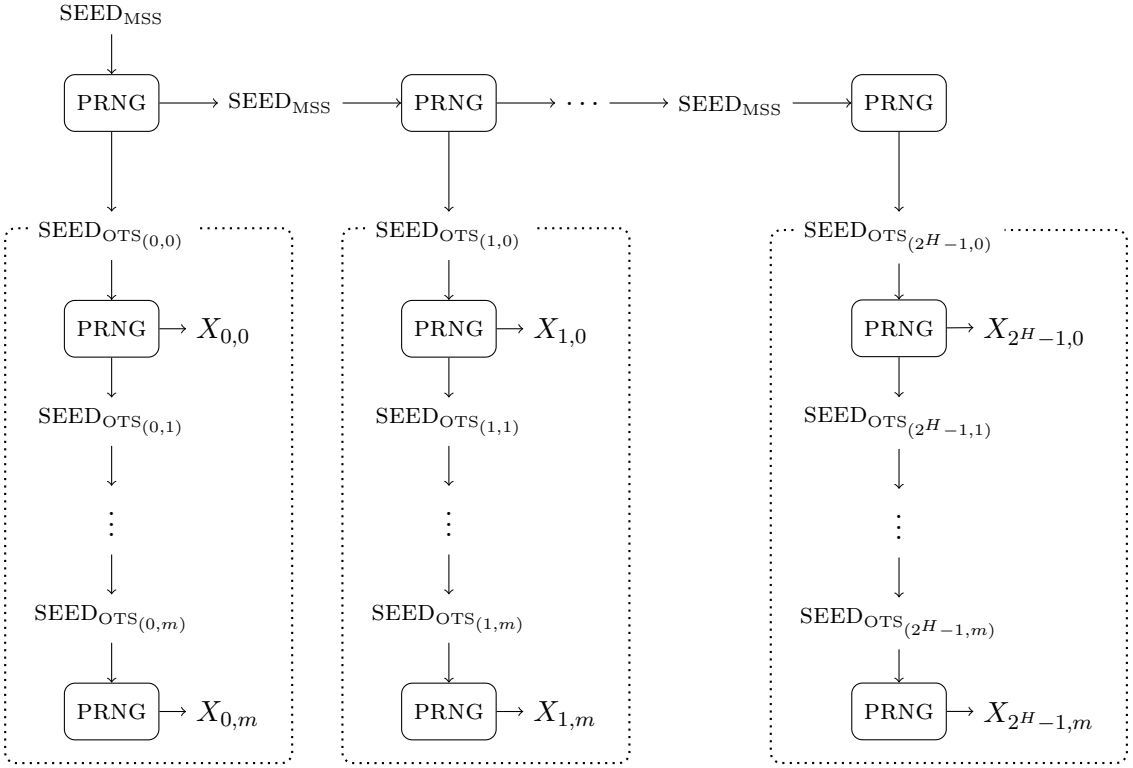



**Figure 3.4:** One-time signature secret key generation using a pseudorandom number generation in the context of a Merkle tree authentication scheme. The dotted boxes represent the different instances of one-time signatures. [BDS09]

The public key still consists of the root of the Merkle tree.

$$Y \leftarrow \nu_H[0].$$

The signing process is similar to the original procedure, except that the PRNG is now used with a secret seed to generate the OTS secret values. $\text{SEED}_{\text{MSS}}$ is updated after each use, so the scheme is always ready to sign the next message. With this change, the scheme achieves *forward secrecy*, since previous seeds cannot be recovered from future seeds. This means that even after a key revocation, the signatures that were previously issued remain valid. The verification process is unaffected by the changes.

### 3.4.2 Goldreich's stateless improvement

To conform to modern applications, the statefulness of Merkle's authentication scheme needs to be eliminated. Oded Goldreich addressed this issue by creating a scheme which achieves complete statelessness [Gol04]. Thanks to this improvement, the signer is not required to remember the one-time signature instances that have already been used. Actually, each possible digest is bound to a unique signature which is computed on the fly. This makes the scheme vulnerable only to collision attacks of the digest computation. It consists of constructing a Merkle signature scheme in which each node in the tree is a different one-time signature instance, so that the leaves can be selected according to the message digest.

In the original MSS only the leaves were OTS instances, while the nodes were hashes of the two children concatenated. Here, in the Goldreich cryptosystem, every node of the tree is constructed using a one-time signature, so the signer never needs to compute the whole tree at once. The leaves are still used to sign the message digest, while the non-leaf nodes are used to sign the concatenation of the two verification keys from its two children. This reduces the key generation time, since the signer can only publish the verification key of the OTS instance at the root of the tree. However, this substantially increases the signature and key sizes, as well as the complexity of the signing and verifying procedures, since many more one-time signatures are involved.

The parameters are the same as those used for the Merkle signature scheme.

Figure 3.5 depicts a small example of the Goldreich digital signature scheme. In this case, the authentication tree is of height $H = 3$. Each node consists of an OTS instance, with respect to their corresponding key pair $X_i$ and $Y_i$. The hash value of the message $M$ to be signed decides the leaf to use. The full signature contains all the one-time signatures across the tree, from the leaf to the root, in addition to the verification keys from the siblings nodes.

As one can observe in the above figure, every possible digest is bound to a particular leaf, meaning that a digest of size $m$ requires $2^m$ leaves, hence a height of $H = m$. Because of the birthday paradox, it also means that the security cannot be greater than $m/2$ bits.

With these changes, the overall tree includes a total of $2^{H+1} - 1$ OTS instances. The scheme requires as many OTS secret keys as this amount[2]. The public key, however, consists only of the verification key from the top one-time signature instance:

$$X \leftarrow \begin{pmatrix} X_{\mathrm{OTS}_0} \\ X_{\mathrm{OTS}_1} \\ \vdots \\ X_{\mathrm{OTS}_{2^{H+1}-1}} \end{pmatrix}, \qquad Y \leftarrow Y_{2^{H+1}-1}.$$

The signing procedure starts by retrieving the leaf index by using the value of the message digest $d = h(M)$. The one-time signature associated to this leaf index is used to produce

[2]The original scheme already makes use of the pseudorandom number generator improvement, so it can only be a seed.

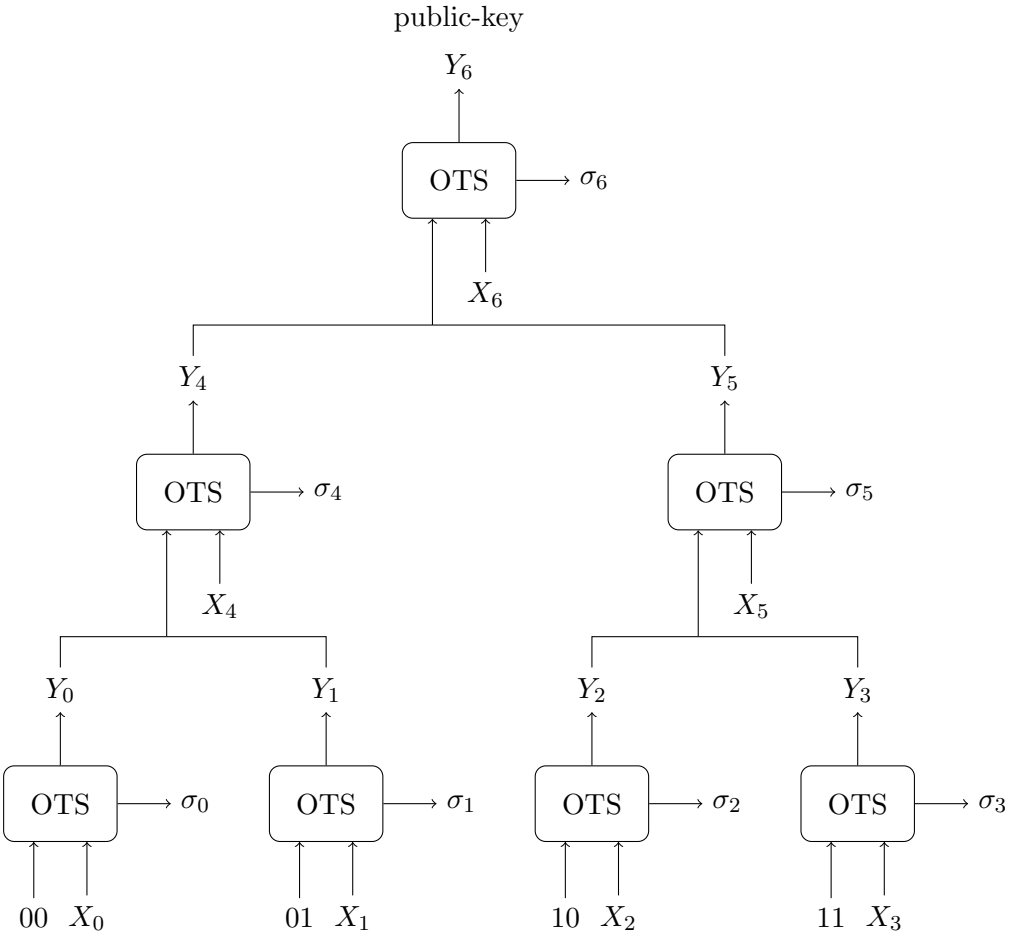



**Figure 3.5:** Example of an instance of Goldreich's digital signatures scheme.

the digest signature $\sigma_d$ and the verification key[3] $Y_d$. The sibling leaf verification key $Y_{d\pm1}$ also needs to be generated, so the concatenation of the two verification keys can be signed with the parent instance. This process of generating the verification key of the sibling node is repeated across the entire tree, up until the root. In total, the whole procedure amounts to $2H$ verification key generations, and $H$ one-time signatures.

Verifying the correctness of a signature requires verifying each one-time signature included in the signature, in addition to using the overall public key $Y$ to verify the authenticity of the two last verification keys. This procedure amounts to a total of $H$ OTS verifications.

A variant of Goldreich's cryptosystem, suggested by the author himself, consists of making the leaf selection randomly, rather than choosing the leaf deterministically. This still makes the scheme stateless, but probabilistic, which allows the height of the tree to be a bit less than the size of the digest and still avoids accidental leaf collision.

[3]The verification key can be generated with either the OTS key $X_d$, or the pseudorandom number generator fed with the secret seed.

### 3.4.3 Chaining trees

Besides being stateful, the other main downside of Merkle digital signature scheme was the key generation and signature procedure being exponential in the height of the tree. To solve this issue, Johannes Buchmann et al. proposed an abstraction of the scheme by using multiple instances of the Merkle tree authentication scheme to construct a whole new tree [Buc+06]. The design results in a hypertree, i.e., a tree made of trees, whose upper subtrees authenticate lower subtrees. The method is known as the tree chaining method and referred to as CMSS[4].

Goldreich's digital signature scheme can actually be seen as a stateless variant of CMSS with subtrees of height zero. Indeed, by annexing a Merkle tree on top of the OTS instances at the nodes of Goldreich's cryptosystem, we end up with CMSS. Combining the two becomes therefore natural, as done with SPHINCS.

This new abstraction requires adjustments on the Merkle's tree authentication scheme parameters and definitions:

- $d \in \mathbb{N}$ : the number of layers of subtrees
- $H = k \cdot d$ : the height of the whole hypertree, where $k \in \mathbb{N}$
- $\nu_i^{(l,j)}[k]$ : the $k^{\text{th}}$ node of altitude $i$ in the $j^{\text{th}}$ subtree on layer $l$.

An instance of the CMSS scheme is illustrated in Figure 3.6. Here, the triangles consist of sub-trees, which themselves form a hypertree. The arrows represent the available leaves of the subtrees that can be used to authenticate either messages or roots of lower subtrees.

In this particular example, there are $d = 3$ layers of subtrees, and the height of the hypertree is $H = 3$, leading to $2^H = 8$ signatures available. The height of each subtree is $H/d = 1$, making the key generation light, since it consists of only computing the root of the highest tree which possesses only 3 nodes. However, the full signature of a message will include the different paths across all the subtrees, in addition to the one-time signatures from their leaves.

The key generation requires to first generate the keys of the OTS instances at the leaves of the top subtree. It then computes the root of this subtree, which corresponds to the overall public key of the cryptosystem. The secret key, however, includes the secret keys of all the OTS instances[5]:

$$X \leftarrow \begin{pmatrix} X_{\text{OTS}_0} \\ X_{\text{OTS}_1} \\ \vdots \\ X_{\text{OTS}_T} \end{pmatrix}, \qquad Y \leftarrow \nu_{H/d}^{(d-1,0)}[0].$$

where $T = \sum_{i=0}^{d} 2^{i \cdot H/d} - 1 = \frac{2^{H/d}(2^H - 1)}{2^{H/d} - 1}$ is the number of OTS secret keys the scheme requires.

[4]We suspect that CMSS stands for "Coronado–Merkle Signature Scheme", since the full name was never mentioned in the original paper.

[5]As in Goldreich's scheme, the original CMSS already includes the pseudorandom number generator improvement, implying that it suffices to store a single seed value.

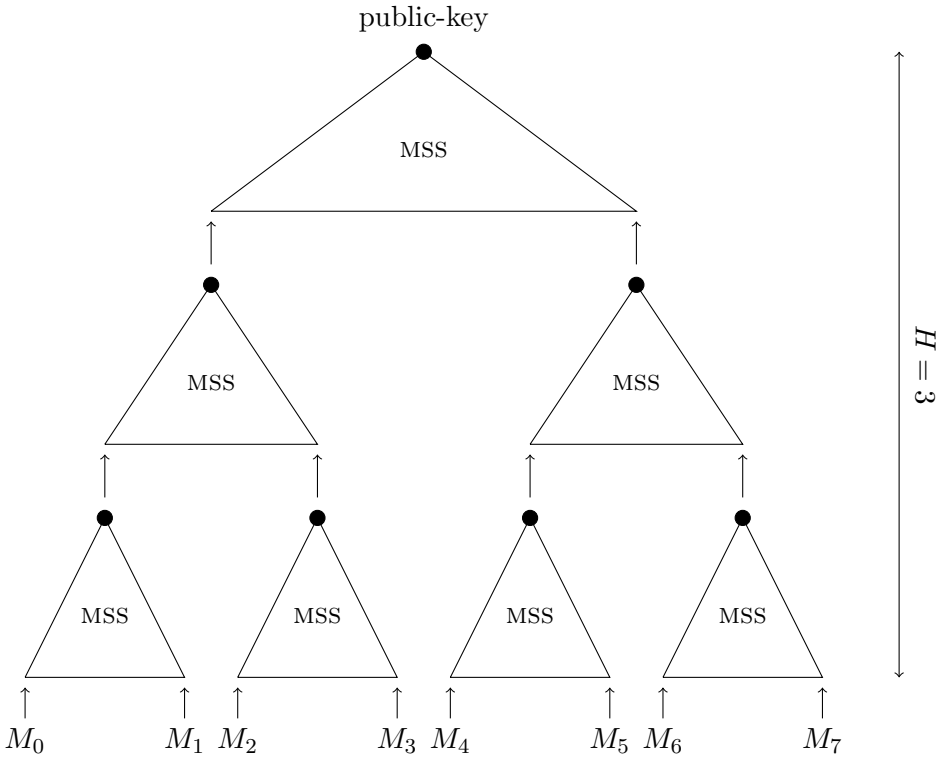



**Figure 3.6:** Example of an instance of the tree chaining method as in CMSS. The hypertree is of height $H = 3$ and consists of $d = 3$ layers of subtrees.

Generating the public key requires a total of $2^{H/d+1} - 1$ nodes (the top subtree) to be computed, compared to the $2^{H+1} - 1$ nodes that were initially needed in the original Merkle signature scheme. This is at the cost of increasing the secret key size by a factor of $(2^H - 1)/(2^d(2^{H/d} - 1))$ (where $H \neq 0$). CMSS is also still stateful: the signer still needs to remember all the instances that have been used.

The signing procedure consists of chaining Merkle scheme signing procedures together in such a way that it makes a path across the subtrees from the leaf up to the top root. To achieve this, upper MSSs sign the Merkle tree root of lower MSSs whose signatures are added to the final signature. The subtree at the bottom of the scheme works exactly as the original MSS. In the end, the full signature consists of all the one-time signatures from all the traversed subtrees, as well as the authentication path from the MSSs themselves. This amounts to a total of $d \cdot (2^{H/d+1} - 1)$ nodes to compute, since the subtrees that are not in the path are omitted, in addition to $d$ OTS signing procedures to make. The authentication path still includes $H$ nodes, along with the $d$ one-time signatures.

As one can expect, the verification procedure re-creates each subtree in the path from the leaf to the root, and verifies the correctness of their roots with the provided one-time signatures. Finally, the root of the top subtree is compared to the public key.

### 3.4.4 Second-preimage resistance

The fact that a hash-based signature scheme should be determined by the second-preimage resistance property of the underlying hash function rather than its collision resistance was discussed for a long time (see [NSW05]). It was resolved by Erik Dahmen et al. in 2008 when they proposed a new construction for Merkle's tree authentication scheme—referred to as SPR-MSS, i.e., "Second-Preimage-Resistant Merkle Signature Scheme"—which was proven secure assuming only a second-preimage resistant hash function [Dah+08]. This resulted in a cryptosystem which was not affected by collision finding speed-ups, such as the birthday paradox or Grover's algorithm. The idea was inspired by the XOR tree from Mihir Bellare and Phillip Rogaway [BR97] which consists of applying a bitwise XOR operation with a different bitmask to each node computation.

Adding a layer of XOR operations to transform a collision-resistant scheme to a second-preimage resistant one is a technique that can actually be applied to any other hash-based schemes, as we will see with W-OTS$^+$. In our case, the technique consists of modifying the computation of the Merkle tree according to the following definition:

**Definition 3.2.** (SPR-Merkle tree) An *SPR-Merkle tree* of height $H \in \mathbb{N}$ is a binary tree $\nu$ whose nodes $\nu_i$ at altitude $1 \leq i \leq H$ and index $0 \leq j \leq 2^{H-i}$ are defined as follows:

$$\nu_i[j] = g\Big(\big(\nu_{i-1}[2j] \oplus r_{i-1}[2j]\big) \,||\, \big(\nu_{i-1}[2j+1] \oplus r_{i-1}[2j+1]\big)\Big)$$

where $g : \{0,1\}^{2n} \to \{0,1\}^n$ is a random one-way compression function of security parameter $n \in \mathbb{N}$ and $r_i[j] \in \{0,1\}^n$ a unique bitmask.

An illustration of a small SPR-Merkle tree of height $H = 2$ is shown in Figure 3.7. The leaves represent the hashed versions of the $2^H = 4$ verification keys of OTS, which are then XORed with a corresponding bitmask. The result is then hashed with the one-way compression function $g$ and the process repeats itself up to the root $\nu_2[0]$.

The bitmasks require $2(2^H - 1)$ random values for masking each input, i.e., one bitmask per node, except for the root, and need to be generated during the key generation process. The array of bitmasks $r$ is therefore created as follows:

$$r_i[j] \sim \mathcal{U}(\{0,1\}^n) \quad \text{for } 0 \leq i \leq H-1,\ 0 \leq j \leq 2^{H-i}.$$

Since they need to be remembered for the signing procedure, the bitmasks are included in the secret key $X$, which, otherwise, still consists of the one-time signature secret keys:

$$X \leftarrow \begin{pmatrix} r \\ X_0 \\ \vdots \\ X_{2^H-1} \end{pmatrix}.$$

The public key $Y$ will also contain the bitmasks, in addition to the root of the SPR-Merkle tree, since they are also necessary for the verification procedure:

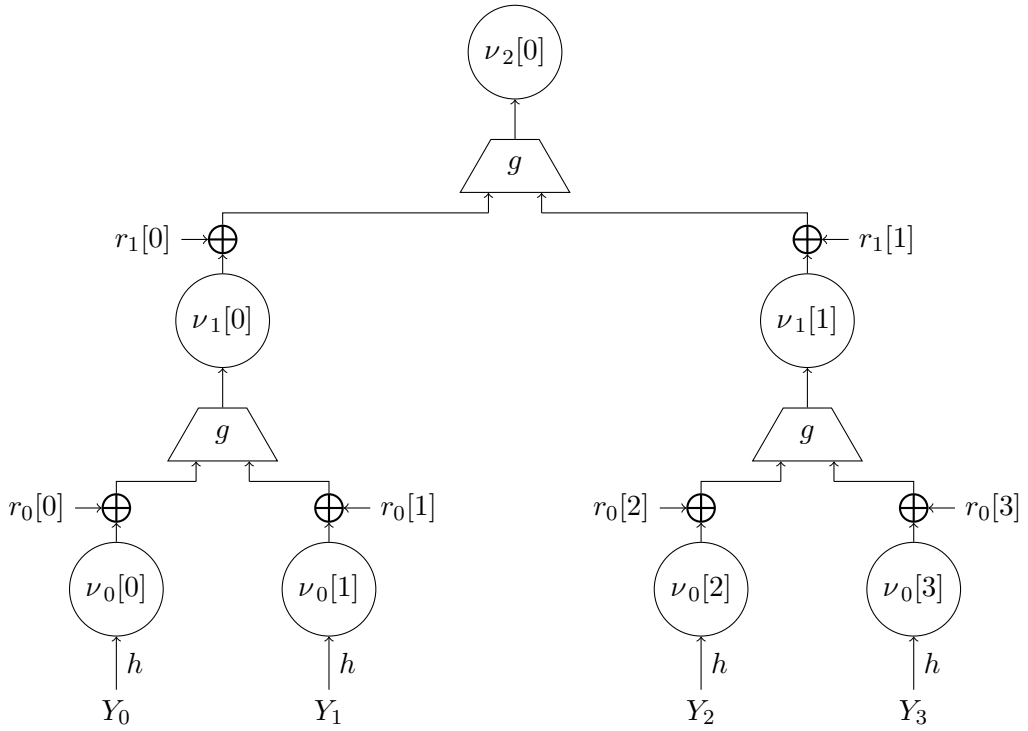



**Figure 3.7:** Example of an SPR-Merkle tree of height $H = 2$.

$$Y \leftarrow \begin{pmatrix} r \\ \nu_H[0] \end{pmatrix}.$$

Aside from XORing the input of each random one-way compression function with the corresponding bitmask, the signing and verification procedures are the same as in MSS.

### 3.4.5 W-OTS$^+$

In 2011, Johannes Buchmann et al. investigated the security of the Winternitz one-time signature scheme [Buc+11]. In particular, they found that the scheme is existentially unforgeable under adaptive chosen message attacks when a family of pseudorandom functions was used in place of a collision-resistant random one-way function. This allows the usage of a hash function with shorter output to be used, since, in this case, the birthday paradox is avoided, resulting in a W-OTS variant with shorter signature size.

This substitution consists of replacing the random one-way function $f$ by the following:

- $F(n) = \left\{ f_k : \{0,1\}^n \to \{0,1\}^n \mid k \in \{0,1\}^n \right\}$ : a family of pseudorandom functions.

Based on this update, Andreas Hülsing presented another Winternitz-type of one-time signature scheme which adds a layer of XOR operations on top of the pseudorandom functions input to reduce its security to second-preimage resistance. This scheme is known as *W-OTS$^+$* [Hül13] and produces even shorter signatures. The idea is to simply XOR the input of every PRF with a set of random but known bitmasks.

This additional layer is characterized by a similar but different chaining method, since it now takes a list of randomly generated bitmasks as a new parameter. Let $x \in \{0,1\}^n$ denote the starting point of the chain, $i \geq 0$ the length of the chain, $r_j \in \{0,1\}^n$ the bitmasks for $0 \leq j < i$, and $k \in \{0,1\}^n$ the PRF key. The chaining is defined as follows:

$$c_k^i(x,r) = \begin{cases} x & \text{if } i = 0 \\ f_k(c_k^{i-1}(x,r) \oplus r_{i-1}) & \text{if } i > 0. \end{cases}$$

The key for the pseudorandom functions family is fixed and joined to the public key, while their input consists of the values from the secret key. In addition to this, key generation requires $W-1$ more values to be randomly drawn. These will serve as bitmasks to the input of the PRF:

$$r \leftarrow \begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_{W-2} \end{pmatrix}$$

where $r_i \sim \mathcal{U}(\{0,1\}^n)$ for $0 \leq i < W$.

The secret key $X$ and public key $Y$ are therefore generated as follows:

$$X \leftarrow \begin{pmatrix} r \\ x_0 \\ \vdots \\ x_{\ell-1} \end{pmatrix}, \quad Y \leftarrow \begin{pmatrix} k \\ r \\ c_k^{W-1}(x_0,r) \\ \vdots \\ c_k^{W-1}(x_{\ell-1},r) \end{pmatrix}$$

where $x_i \sim \mathcal{U}(\{0,1\}^n)$ for $0 \leq i < \ell$, and $k \sim \mathcal{U}(\{0,1\}^n)$.

In the context of SPHINCS, the signing phase for W-OTS$^+$ is the same as the original Winternitz one-time signature scheme, except that the verification key is not included in the full signature. Instead, the verifier will derive the verification key from the one-time signature, and rather than comparing it to a public key, it will be authenticated later using the SPHINCS verification algorithm. In this case, the W-OTS$^+$ verification procedure consists of retrieving what the verifier believes to be the OTS verification key.

### 3.4.6 HORST

One way to have a better few-times signatures scheme than HORS is to combine the design of said scheme with the construction of a Merkle tree, as it is done with one-time signatures in Merkle's tree authentication scheme. This leads to the "HORS with Trees" scheme (HORST), which was first mentioned for the SPHINCS construction [Ber+14]. The idea is to construct a Merkle tree starting with the initial HORS verification keys, so the public key now consists only of the root of the resulting tree. This produces a memory-time tradeoff between the generation of the signatures and the size of the public key and signatures.

The HORST scheme is illustrated in Figure 3.8. Here, there are $k = 4$ different blocks of size $\tau = 2$. The leaves of the tree correspond to the hashed version of the secret values $x_i$. The structure of the nodes follows the definition of a Merkle tree, up to the root $\nu_2[0]$ which corresponds to the public-key of the scheme.

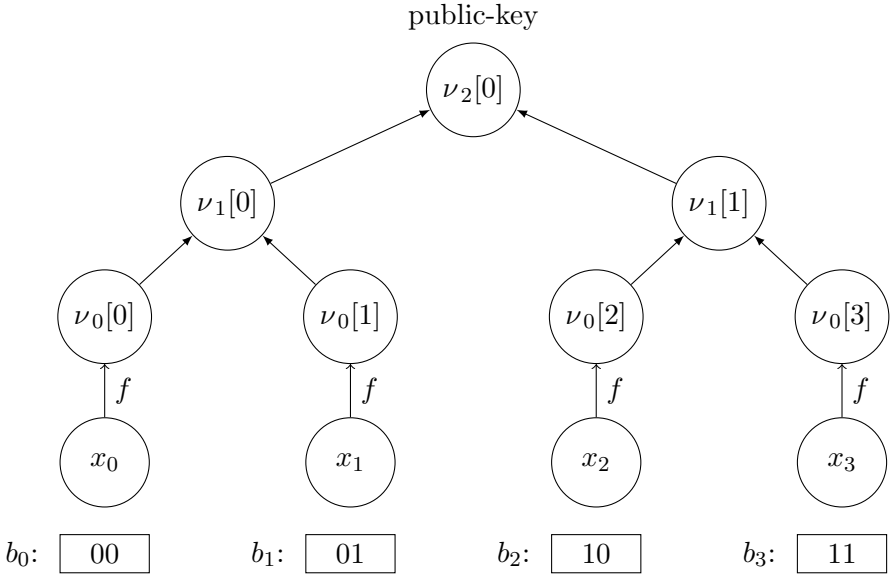



**Figure 3.8:** Example of HORST.

To further improve the scheme, a layer of XOR operations can be added to the computation of each node. This causes the scheme to require only the hash function to be second-preimage resistant, as for SPR-MSS.

In order to put HORST into practice, we start with a HORST instance and add a function to compute the nodes of the Merkle tree to the parameters:

- $g : \{0,1\}^{2n} \rightarrow \{0,1\}^n$ : a random one-way compression function.

As in the HORS scheme, the signer needs to draw $t = 2^\tau$ random secret values which will then be hashed using the one-way function. The result will be combined with the construction of a Merkle tree of height $\tau$, as done in Algorithm 3.2.1.

The private key $X$ consists of the random secret values:

$$X \leftarrow \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{t-1} \end{pmatrix}$$

where $x_i \sim \mathcal{U}(\{0,1\}^n)$ for $0 \leq i < t$.

The public key $Y$ consists of the root of the Merkle tree:

$$Y \leftarrow \nu_\tau[0].$$

As in HORS, the signing procedure consists of revealing the secret values corresponding to each block of the message digest, but the signer also needs to add the necessary nodes in the Merkle tree to allow a verifier to compute its root. This results in as many authentication paths as there are secret values. Since it is likely that those paths share the same nodes (especially on higher height), the signing process can also stop at a certain altitude and always output the nodes at this height, whether they belong to any of the authentication paths or not. This results in a truncated Merkle tree whose "tree buds" are now used to verify the secret values with the authentication path i.e., as if they were the roots of multiple Merkle trees, and re-create the public key.

### 3.4.7 L-Tree compression

An L-Tree—an abbreviation for a Lindenmayer Tree—is an unbalanced binary hash tree which allows the compression of multiple values to a single value, in a similar way as in a Merkle tree. The term first appeared in the SPHINCS paper, but the concept was borrowed from Erik Dahmen's paper [Dah+08]. Formally, the construction of an L-Tree consists of constructing a binary hash tree starting from any amount of leaves just like a Merkle tree but with one additional rule: if a node does not have a right sibling, it is lifted up to higher levels until it becomes a right sibling itself.

Figure 3.9 shows a short example of how to obtain an L-Tree starting from five values. As one can see, the number of leaves is not a power of two which necessary leads to an unbalanced tree. Here, the fifth leaf $\nu_0[4]$ has no right sibling, therefore its level must increase until it reaches a layer where it is considered as a right sibling. To achieve this, its altitude needs to be increased by two levels, so it can be compressed with the result of the left binary tree.

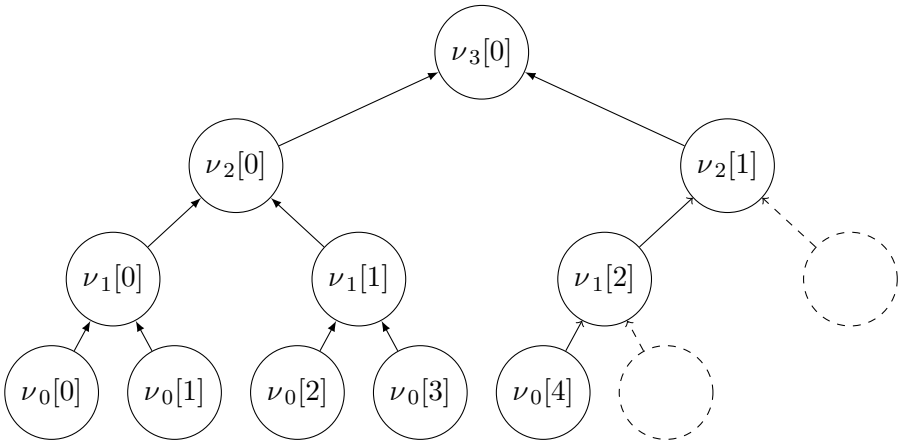



**Figure 3.9:** Example of an L-Tree with five leaves. The leaf $\nu_0[4]$ having no right sibling, it is elevated up to the second layer where it is the right sibling of $\nu_2[0]$.

The node computation works as in Merkle trees and can use the same random one-way compression function $g$. Therefore, the previous improvements on the Merkle signature scheme, such as SPR-Merkle trees, can also be applied to L-Trees. The height of an L-Tree with $N$ leaves is $\lceil \log N \rceil$. It is used in SPHINCS especially to compress the values from the W-OTS$^+$ verification key to a single value of size $n$; the root of the L-Tree.

### 3.4.8 SPHINCS construction

The construction of the SPHINCS cryptosystem involves the combination of all the aforementioned improvements, in order to make a practical stateless digital signature scheme based on hash functions. Concretely, it creates a HORST instance depending on the message value to sign its digest, uses a W-OTS$^+$ instance to authenticate the root of the HORST instance, compresses the W-OTS$^+$ verification key using a masked L-Tree, and repeats this process as in CMSS starting from the L-Tree root of the verification key up until the final root of the hypertree is reached.

In order to save memory, the scheme makes use of a pseudorandom number generator to generate HORST and W-OTS$^+$ secret keys on the fly. The PRNG seed is deterministically drawn using a family of pseudorandom functions. Another ensemble of pseudorandom functions is used with the message value and a part of the secret key to select the HORST instance which will sign its digest. Also, a layer of XOR operations with predetermined bitmasks is annexed to the input of each application of the random one-way compression function, so the scheme security essentially relies on the second-preimage resistance of the function.

A short illustration of a SPHINCS instance is shown in Figure 3.10. There are $d = 2$ layers of subtrees of height $H/d = 2$, leading thus to a total height of $H = 4$. The signing process starts at the bottom of the figure (at layer $l = \text{HORST}$) by choosing a HORST instance according to the message a part of the secret key. In this example, the tenth instance ($\text{HORST}_9$) has been chosen to sign the message digest. The HORST signature $\sigma_H$ consists of revealing the secret values ($x_0$, $x_1$, $x_2$) and their corresponding authentication paths in the HORST tree.

The rest of the scheme chains Merkle signature schemes as in CMSS to sign the HORST tree root, starting from the bottom MSS at the index corresponding to the instance number of the selected HORST. In the example, the second W-OTS$^+$ at layer $l = 0$ and index $i = 2$ signs the $\text{HORST}_9$ tree root, producing $\sigma_{\text{W},\,0}$ which will be added to the full signature. The resulting W-OTS$^+$ verification key $Y_{(0,\,9)}$ is then compressed with an L-Tree, whose root actually represents the second leaf of $\text{MSS}_{(0,\,2)}$. The authentication path to $\text{MSS}_{(0,\,2)}$ root is added to the final signature, and the process repeats itself, starting this time from $\text{MSS}_{(0,\,2)}$ root.

In order to address to a specific leaf in a subtree for the pseudorandom generation, a simple addressing scheme is established. An address $A(i,\,j,\,l)$ which refers to the leaf $0 \leq i < 2^{H/d}$ of the subtree $0 \leq j < 2^{(d-1-l)H/d}$ at layer $0 \leq l < d$ is structured as follows:

$$A(i,\,j,\,l) = (l \,||\, j \,||\, i)$$

such that the first $\lceil \log(d+1) \rceil$ bits of $A(i,\,j,\,l)$ correspond to the layer $l$, the next $(d-1)(H/d)$ bits refer to the subtree $j$ at layer $l$, and the last $H/d$ bits represent the leaf $i$. Thus, the bit length of an address amounts to $\lceil \log(d+1) \rceil + (d-1)(H/d) + (H/d) = \lceil \log(d+1) \rceil + H$.

**Setup** A SPHINCS instance of security parameter $n \in \mathbb{N}$ requires a whole list of public parameters which mainly regroups all the parameters from the previous improvements. Let $m = \text{poly}(n)$ be the size of the digest, i.e., the number of bits to sign.

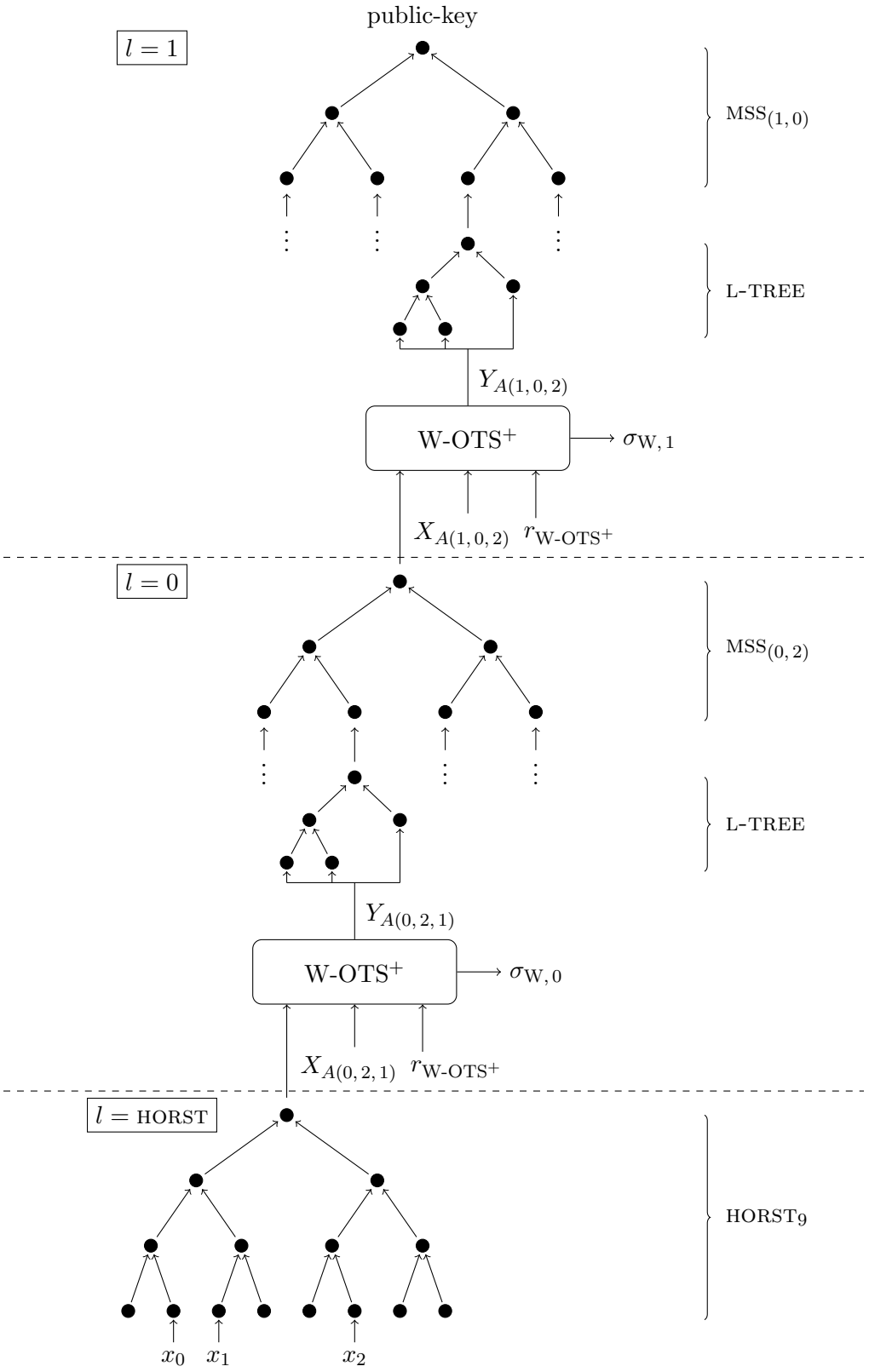



**Figure 3.10:** Example of SPHINCS.

First of all, the scheme involves diverse hash functions:

- $f : \{0,1\}^n \to \{0,1\}^n$ : a random one-way function
- $h : \{0,1\}^n \times \{0,1\}^* \to \{0,1\}^m$ : a cryptographic hash function
- $g : \{0,1\}^{2n} \to \{0,1\}^n$ : a random one-way compression function

as well as various pseudorandom functions:

- $G_\lambda : \{0,1\}^n \to \{0,1\}^{\lambda n}$ : a pseudorandom number generator
- $F_\lambda : \{0,1\}^\lambda \times \{0,1\}^n \to \{0,1\}^n$ : an ensemble of pseudorandom functions
- $F : \{0,1\}^* \times \{0,1\}^n \to \{0,1\}^{2n}$ : a family of pseudorandom functions.

All these functions have been made distinct according to their roles, but the authors note that they can be built from a single cryptographic hash function. Furthermore, the SPHINCS setup needs the following parameters:

- $d \in \mathbb{N}$ : the number of layers of subtrees
- $H \in \mathbb{N}$ : the height of the whole hypertree, such that $H = k \cdot d$ for $k \in \mathbb{N}$
- $w \in \mathbb{N}$ : the W-OTS$^+$ window of bits to sign simultaneously
- $k$ : the HORST number of blocks to sign
- $\tau$ : the HORST number of bits inside one block, where $k \cdot \tau = m$.

Moreover, for ease of notation, the definitions of the following quantities are made:

- $W = 2^w$ : the length of a W-OTS$^+$ chain
- $\ell_1 = \left\lceil \frac{n}{w} \right\rceil$ : the number of bit strings in a W-OTS$^+$ digest
- $\ell_2 = \left\lceil \frac{\lfloor \log \ell_1 \rfloor + 1 + w}{w} \right\rceil$ : the number of bit strings in the W-OTS$^+$ checksum
- $\ell = \ell_1 + \ell_2$ : the total number of bit strings to sign with W-OTS$^+$
- $t = 2^\tau$ : the maximum amount of possible HORST blocks
- $a = \lceil \log(d+1) \rceil + H$ : the leaf addresses bit length.

**Key generation** To generate the overall secret key, the scheme requires to uniformly draw two secret seeds:

$$\text{SK}_1 \sim \mathcal{U}(\{0,1\}^n), \quad \text{SK}_2 \sim \mathcal{U}(\{0,1\}^n).$$

The first one will be used to generate the secret seeds for the different signing components, while the second one will be used to select unpredictable HORST instances.

In addition to the seeds, the secret key generation also needs to set up the bitmasks that will mask each input of the random one-way compression function. The total amount of bitmasks is determined by the biggest amount required by each component (W-OTS$^+$, L-Tree & SPR-MSS, and HORST), since the same set can be used for each of them:

$$p = \max\{W - 1,\, 2(H/d + \lceil \log \ell \rceil),\, 2\tau\}\,.$$

A number of $p$ bitmasks $r$ are therefore drawn:

$$r \leftarrow \begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_{p-1} \end{pmatrix}$$

where $r_i \sim \mathcal{U}(\{0,1\}^n)$ for $0 \leq i < p$.

For the sake of clarity, let $r_{\text{W-OTS}^+}$ correspond to the first $W - 1$ bitmasks of $r$, let $r_{\text{trees}}$ to the first $2(H/d + \lceil \log \ell \rceil)$ bitmasks, and let $r_{\text{HORST}}$ to the first $2\tau$ bitmasks. Furthermore, let $r_{\text{L-Tree}}$ refer to the first $2(H/d)$ bitmasks of $r_{\text{trees}}$, and $r_{\text{MSS}}$ to the next $2\lceil \log \ell \rceil$ bitmasks. The same bitmasks are indeed shared among hash-based instances.

To compute the public key, the top subtree needs to be constructed. This step requires the generation of all the W-OTS$^+$ verification keys at the leaves of the top subtree. These verification keys are computed on the fly by recovering their secret keys with the PRNG fed with their corresponding seed, and by using the bitmasks $r_{\text{W-OTS}^+}$. This seed is retrieved by evaluating the family of PRF $F_\lambda$ where $\lambda = a$ with the address of the leaf and the first secret seed. Then, these verification keys are compressed using an L-Tree which is constructed with the masks $r_{\text{L-Tree}}$ whose roots correspond to the leaves of the Merkle subtree. Once all the leaves have been computed, the entire SPR-Merkle subtree is computed with the masks $r_{\text{MSS}}$, and the root corresponds to the public key.

The SPHINCS public key generation under the secret key $(\text{SK}_1, \text{SK}_2)$ and the bitmasks $r$ is described in Algorithm 3.4.1.

**Signing procedure** The SPHINCS signing procedure can be split into two parts. In the first part, a HORST instance is selected according to the message and the second secret seed to actually sign the digest. In the second part, the root of the HORST tree is authenticated in a CMSS fashion.

To sign a message, the SPHINCS scheme first needs to derive two pseudorandom values $(R_1, R_2)$: one to constitute the actual hash digest, and the other to select the HORST instance that will sign it. This is done by calling the PRF $F$ with the message and the second secret seed $\text{SK}_2$, and by splitting its result in half. The hash digest consists of the result of the hash function $h$ with the first pseudorandom value $R_1$ and the message. The address of the HORST instance $A_H$ is composed of the first $h$ bits of the second pseudorandom value $R_2$ prefixed with the value $d$.

The seed for the generation of the selected HORST secret key will be generated by evaluating the family of PRF $F_\lambda$ where $\lambda = a$ with the address $A_h$ and the first secret seed

**input** : $1^n$ – the security parameter
**input** : $\text{SK} = (\text{SK}_1, \text{SK}_2, r)$ – the overall secret key
**output** : $\text{PK} = (\nu_{H/d}^{(d-1,0)}[0], r)$ – the overall public key

---

**for** $i = 0$ **to** $2^{H/d} - 1$ **do**
  $A \leftarrow A(d-1, 0, i)$
  $\text{SEED}_A \leftarrow F_a(A, \text{SK}_1)$
  $X_A \leftarrow G_\ell(\text{SEED}_A)$
  $Y_A \leftarrow \texttt{KeyGen}_{\texttt{W-OTS}^+}(1^n, r_{\text{W-OTS}^+}, X_A)$
  Let $\nu_0^{(d-1,0)}[i]$ be the L-Tree compression of $Y_A$ with $r_{\text{L-Tree}}$
**end**
Construct the SPR-Merkle tree $\nu_{H/d}^{(d-1,0)}$ with $r_{\text{MSS}}$
$\text{PK} \leftarrow (\nu_{H/d}^{(d-1,0)}[0], r)$
**return** PK

**Algorithm 3.4.1:** SPHINCS public key generation algorithm.

$\text{SK}_1$. This seed will generate $t$ secret values when fed to the PRNG, so the HORST signature of the digest $\sigma_H$ can be computed as well as the verification key $Y_H$. The HORST signature is then output, but not the verification key, since it is left to the verifier to compute it from $\sigma_H$. It is however held in memory for the next part which will undertake its authentication.

The rest of the procedure repeatedly signs the previously computed root with the Merkle scheme on the next layer at a specific address $A_l$ derived from the previous address: the $H/d$ last bits of the last subtree index become the new leaf index, while the remaining bits give the new subtree index. As in the key generation, all the W-OTS$^+$ verification keys $Y_{W,l}$ at the leaves of the subtree on the $l^{\text{th}}$ layer need to be generated and compressed to the root of an L-Tree. This is done by recovering their corresponding secret keys $X_{W,l}$ with the PRNG using, as a seed, the result of the PRF $F_\lambda$ where $\lambda = a$ at their leaf address $A_l$ and the first secret seed. Moreover, the W-OTS$^+$ responsible to sign the root will output the respective signature $\sigma_{W,l}$, but the verification key $Y_{W,l}$ is omitted since it can be deduced by the verifier. The authentication path from the leaf index to the root $\text{Auth}_l$ is also output, and the root is updated with the current Merkle tree.

The full signature is therefore: $\Sigma = (\text{INDEX}, R_1, \sigma_H, \text{Auth}_H, \sigma_{W,0}, \text{Auth}_0, \ldots, \sigma_{W,d-1}, \text{Auth}_{d-1})$.

The SPHINCS signing procedure of a message $M \in \{0,1\}^*$ under the secret key $(\text{SK}_1, \text{SK}_2)$ and the bitmasks $r$ is described in Algorithm 3.4.2.

**Verification procedure** The SPHINCS verification procedure works similar to the signature procedure. First, the digest is computed and the HORST verification key is retrieved using the corresponding signature. Then, the authenticity of the recovered HORST root is verified with the re-construction of the top layer in the CMSS hypertree.

**input :** $M$ – the message
**input :** $\text{SK} = (\text{SK}_1, \text{SK}_2, r)$ – the overall secret key
**output :** $\Sigma = (\text{INDEX}, R_1, \sigma_H, \text{Auth}_H, \sigma_{W,0}, \text{Auth}_0, \ldots, \sigma_{W,d-1}, \text{Auth}_{d-1})$ – the full SPHINCS signature

---

$(R_1, R_2) \leftarrow F(M, \text{SK}_2)$
$D \leftarrow h(M, R_1)$
Let $j$ be the first $(d-1)H/d$ bits of $R_2$, and $i$ the next $H/d$ bits
$\text{INDEX} \leftarrow (j \,||\, i)$
$A \leftarrow A(d, j, i)$
$\text{SEED}_H \leftarrow F_a(A, \text{SK}_1)$
$X_H \leftarrow G_t(\text{SEED}_H)$
$Y_H \leftarrow \texttt{KeyGen}_{\texttt{HORST}}(1^n, r_{\text{HORST}}, X_H)$
$(\sigma_H, \text{Auth}_H) \leftarrow \texttt{Sign}_{\texttt{HORST}}(D, X_H, r_{\text{HORST}})$
$\text{ROOT} \leftarrow Y_H$
**for** $l = 0$ **to** $d-1$ **do**
  **for** $k = 0$ **to** $2^{H/d} - 1$ **do**
    $A \leftarrow A(l, j, k)$
    $\text{SEED}_A \leftarrow F_a(A, \text{SK}_1)$
    $X_A \leftarrow G_\ell(\text{SEED}_A)$
    $Y_A \leftarrow \texttt{KeyGen}_{\texttt{W-OTS}^+}(1^n, r_{\text{W-OTS}^+}, X_A)$
    **if** $k = i$ **then**
      $\sigma_{W,k} \leftarrow \texttt{Sign}_{\texttt{W-OTS}^+}(\text{ROOT}, X_A, r_{\text{W-OTS}^+})$
    **end**
    Let $\nu_0^{(l,j)}[k]$ be the L-Tree compression of $Y_A$ with $r_{\text{L-Tree}}$
  **end**
  Construct the SPR-Merkle tree $\nu^{(l,j)}$ with $r_{\text{MSS}}$
  **for** $k = 0$ **to** $H/d - 2$ **do**
    $$a_k \leftarrow \begin{cases} \nu_k^{(l,j)}[i/2^k - 1] & \text{if } \lfloor i/2^k \rfloor \equiv 1 \pmod 2 \\ \nu_k^{(l,j)}[i/2^k + 1] & \text{if } \lfloor i/2^k \rfloor \equiv 0 \pmod 2 \end{cases}$$
  **end**
  $\text{Auth}_l \leftarrow (a_0, \ldots, a_{2^{H/d}-2})$
  Update $i$ with the last $H/d$ bits of $j$, and replace $j$ by $\lfloor j/2^{H/d} \rfloor$
  $A \leftarrow A(d-1, j, i)$
  $\text{ROOT} \leftarrow \nu_{H/d}^{(l,j)}[0]$
**end**
$\Sigma \leftarrow (\text{INDEX}, R_1, \sigma_H, \text{Auth}_H, \sigma_{W,0}, \text{Auth}_0, \ldots, \sigma_{W,d-1}, \text{Auth}_{d-1})$
**return** $\Sigma$

**Algorithm 3.4.2:** SPHINCS signing procedure.

To verify the correctness of a message, the hash digest first needs to be computed by evaluating the hash function $h$ with $R_1$ and the message. This digest will be authenticated with the overall SPHINCS procedure which starts by using the HORST signature $\sigma_H$ and $\text{Auth}_H$, as well as the bitmasks $r_{\text{HORST}}$ from the public key, to recover the root of the HORST tree $Y_H$. The INDEX allows the retrieval of the subtree index $j$ and the leaf index $i$ of the HORST tree $Y_H$ in the first Merkle tree.

The rest repeatedly uses the value of the last root computed to recover the W-OTS$^+$ verification key $Y_{W,l}$ of the next layer. The values from the key are compressed to the root of an L-Tree which forms the $i^{\text{th}}$ leaf of the Merkle tree. Thanks to the authentication path $\text{Auth}_l$, the Merkle tree root can be computed. The leaf address is updated as in the signing procedure, i.e., the new leaf index $i$ is given by the last $H/d$ bits of the last subtree index, while the new subtree index $j$ takes the remaining bits of the last subtree index. The procedure repeats itself until it reaches the top layer, so the last root can be compared to the value in the public key.

Note that since no verification key is included in the SPHINCS full signature, the HORST and W-OTS$^+$ verification procedures cannot be used to directly verify the correctness of their respective signatures. These procedures need to be adapted such that they output a presumed verification key that will be used as if it were correct. It is clear that if anything goes wrong at any point of the whole procedure, the error should directly propagate up until the comparison of the top layer root with the value in the public key.

The SPHINCS verification procedure for a signature $\Sigma = (\text{INDEX}, R_1, \sigma_H, \text{Auth}_H, \sigma_{W,0}, \text{Auth}_0, \ldots, \sigma_{W,d-1}, \text{Auth}_{d-1})$ of a message $M \in \{0,1\}^*$ under the public key $\text{PK} = (\nu_{H/d}^{(d-1,0)}[0], r)$ is described in Algorithm 3.4.3.

**Security**

By the principle of the weakest link, the overall security of SPHINCS corresponds to the security of the weakest component. Therefore, every component requires to be tuned such that its security is the same as every other one. Assuming that there are no better attacks than generic attacks, the following addresses the complexity of each individual component in both traditional and quantum settings [Ber+14].

- **Digest second-preimage resistance** : Given $(R, M) \in \{0,1\}^n \times \{0,1\}^*$, finding $(R', M') \neq (R, M)$ such that $h(R', M') = h(R, M)$ costs $2^m$ in traditional settings and $2^{m/2}$ in quantum settings (Grover's algorithm).
- **One-way function first-preimage resistance** : Given an output of $f(x)$, finding $x$ by reverting the one-way function costs $2^n$ in traditional settings and $2^{n/2}$ in quantum settings (Grover's algorithm). Performing a multi-target attack on $T$ hashed values decreases the complexity by the same factor. Since $T \leq 2^H$, this reduces the cost to at most $2^{n-H}$ in traditional settings. In post-quantum settings, however, because overhead occurs when $T > 2^{n/3}$, the number of quantum queries can decrease to $2^{n/2}/\sqrt{T}$, but the required cost would still be $2^{n/2}$ [Ber09]. Note that the number of targets $T$ actually depends on the number of signatures.
- **Subset-resilience** : Given $(R, M) \in \{0,1\}^n \times \{0,1\}^*$, finding $(R', M') \neq (R, M)$ such that $h(R', M')$ is a permutation of the $k$ blocks of bit length $\tau$ from $h(R', M')$ costs no more than $2^m/k!$ in traditional settings. This supposes that the $k$ blocks in

**input** : $M$ – the message
**input** : $\text{PK} = (\nu_{H/d}^{(d-1,\,0)}[0],\, r)$ – the public key
**input** : $\Sigma = (\text{INDEX},\, R_1,\, \sigma_H,\, \text{Auth}_H,\, \sigma_{W,0},\, \text{Auth}_0,\, \ldots,\, \sigma_{W,d-1},\, \text{Auth}_{d-1})$ – the full SPHINCS signature
**output** : `True` if the signature is correct, `False` otherwise

---

$D \leftarrow h(M,\, R_1)$
Let $j$ be the first $(d-1)H/d$ bits of INDEX, and $i$ the last $H/d$ bits
$Y_H \leftarrow \texttt{Verif}_{\texttt{HORST}}(D,\, \sigma_H,\, \text{Auth}_H,\, r_{\text{HORST}})$
$\text{ROOT} \leftarrow Y_H$
**for** $l = 0$ **to** $d-1$ **do**
  $Y_{W,l} \leftarrow \texttt{Verif}_{\texttt{W-OTS}^+}(\text{ROOT},\, \sigma_{W,l},\, r_{\text{W-OTS}^+})$
  Let $R$ be the L-Tree compression of $Y_{W,l}$ with $r_{\text{L-Tree}}$
  Let $(a_0,\, a_1,\, \ldots,\, a_{H/d-2})$ denote the nodes inside $\text{Auth}_l$.
  **for** $k = 0$ **to** $H/d - 2$ **do**
    $$R \leftarrow \begin{cases} g\big((a_k \oplus r_{\text{MSS}}[2k]) \,||\, (R \oplus r_{\text{MSS}}[2k+1])\big) & \text{if } \lfloor i/2^k \rfloor \equiv 1 \pmod 2 \\ g\big((R \oplus r_{\text{MSS}}[2k]) \,||\, (a_k \oplus r_{\text{MSS}}[2k+1])\big) & \text{if } \lfloor i/2^k \rfloor \equiv 0 \pmod 2 \end{cases}$$
  **end**
  $\text{ROOT} \leftarrow R$
  Update $i$ with the last $H/d$ bits of $j$, and replace $j$ by $\lfloor j/2^{H/d} \rfloor$
**end**
**return** $\text{ROOT} \stackrel{?}{=} \nu_{H/d}^{(d-1,\,0)}[0]$

**Algorithm 3.4.3:** SPHINCS verification procedure.

the $h(R',\, M')$ are unlikely to repeat, which is essentially the case when $k^2 << 2^\tau$. In quantum settings, this costs is reduced to $2^{m/2}/\sqrt{k!}$ (Grover's algorithm).

- **$\gamma$-subset-resilience** Given $\gamma$ valid HORST signatures $\sigma_H$ of the same instance, finding $(R',\, M') \neq (R_i,\, M_i)$ for $0 \leq i < \gamma$ so that the set of $k$ blocks of bit length $\tau$ within $h(R',\, M')$ is a subset of the union of the known blocks in the valid $\gamma$ signatures costs $2^{\tau k}/(\gamma k)^k$ in traditional settings. This is because the size of the union is at most $\gamma k$, so a block has a probability of about $\gamma k/2^\tau$ to be uniformly drawn from the set. The $k$ blocks are then supposed to be independent. The cost decreases to $2^{\tau k/2}/(\gamma k)^{k/2}$ in quantum settings (Grover's algorithm).

  Moreover, this resilience is only effective if the attacker has access to $\gamma$ different signatures from the same instance. Assume the SPHINCS instance is used $q$ times, the probability of exactly reusing a specific HORST instance $\gamma$ times is exactly $\binom{q}{\gamma}(1 - 1/2^H)^{q-\gamma}(1/2^H)^\gamma$, which is bounded by $(q/2^H)^\gamma$. The ratio $q/2^H$ defines an important parameter, as the security degrades faster as it approaches 1. With quantum settings, this cost diminishes to $(2^H/q)^{\gamma/2}$ (Grover's algorithm).

- **Indistinguishability of pseudorandom functions** : Distinguishing the output of $G_\ell$, $F_a$, $F$ from ideal random functions costs no more than a generic target attack of $2^n$ in traditional settings. In quantum settings, this attack costs $2^{n/2}$ (Grover's algorithm).

A summary of the above results in quantum settings is listed in Table 3.1 below. In traditional settings, these security bits can be doubled, since the square-root speed-up is mainly provided by Grover's algorithm. The only exception is the one-way functions first-preimage resistance which can be lowered down to $n - H$ because of multi-target attack.

| **Property** | **Security bits** |
|---|---|
| Digest second preimage resistance | $m/2$ |
| Functions first preimage resistance | $n/2$ |
| Subset-resilience | $m/2 - \log(k!)/2$ |
| $\gamma$-subset-resilience ($q$ signatures) | $k(\log\gamma + \log k - \tau)/2 + \gamma(H - \log q)/2$ |
| Indistinguishability of PRF | $n/2$ |

**Table 3.1:** Summary of the security achieved by each component in quantum settings.

### SPHINCS-256

The SPHINCS authors have suggested a standard instantiation of the SPHINCS scheme that achieves 128-sbit in quantum settings [Ber+14]. This instance is called SPHINCS-256, and the concrete choices for the parameters and functions are listed in Table 3.2. In this table, $\pi_{\text{ChaCha}} : \{0,1\}^{512} \to \{0,1\}^{512}$ denotes the permutation step of ChaCha12, and `C` = `"expand 32-byte state to 64-byte state!"` is an ASCII expansion bit string.

In order to attack such an instance with hardware attacks, a custom SAM3X8E ARM Cortex-M3 ARM implementation of the scheme was developed to be run on an Arduino Due board [Ard17]. Andreas Hülsing's already addressed the development of SPHINCS-256 on an ARMed environment [HRS15] which was taken into consideration in our own implementation. The code can be found at `https://github.com/AymericGenet/SPHINCS-arduinodue` and makes use of reference implementations for the hash functions, such as BLAKE [Aum14] and ChaCha [Ber08].

| Parameters | Value | Meaning |
|---|---|---|
| $n$ | 256 | Bit length of hashes in HORST and W-OTS$^+$ |
| $m$ | 512 | Bit length of message digest |
| $d$ | 12 | Layers of the hyper-tree |
| $H$ | 60 | Height of the hyper-tree |
| $w$ | 4 | W-OTS$^+$ block size parameter |
| $k$ | 32 | HORST number of blocks |
| $\tau$ | 16 | HORST bit length of the blocks |
| **Functions** | | |
| Hash function | $h$ | $h(R,\, M) = \text{BLAKE-512}(R \,\|\|\, M)$ |
| OWF | $f$ | $f(M) = \pi_{\text{ChaCha}}(M \,\|\|\, \texttt{C})[0{:}256]$ |
| OWCF | $g$ | $g(M_1,\, M_2) = \pi_{\text{ChaCha}}(\pi_{\text{ChaCha}}(M_1 \,\|\|\, \texttt{C}) \oplus (M_2 \,\|\|\, 0^{256}))[0{:}256]$ |
| PRF | $F$ | $F(M,\, K) = \text{BLAKE-512}(K \,\|\|\, M)$ |
| PRF | $F_a$ | $F_a(A,\, K) = \text{BLAKE-256}(K \,\|\|\, A)$ |
| PRNG | $G_\ell$ | $G(\text{SEED}) = \text{ChaCha12}_{\text{SEED}}(0)_{0,\ldots,\ell-1}$ |
| **Sizes** | | |
| Signature | $\Sigma$ | 41,000 bytes |
| Public-key | PK | 1,056 bytes |
| Private-key | SK | 1,088 bytes |
| Bit masks | $r$ | 1,024 bytes |
| Addresses | $A$ | 8 bytes |

**Table 3.2:** Parameters and functions for the SPHINCS-256 standard [Ber+14].

# Chapter 4

# Side-channel Attacks

A Side-Channel Attack (SCA) describes a way of exploiting the additional information that a physical cryptosystem unintentionally leaks in order to extract the secrets within. These attacks first require the collection of extra information from a device, known as *leakage*, that is assumed related to secret data. This information is then carefully analyzed such that the secrets in question can be partially or entirely recovered.

The cryptanalysis model for hardware SCA involves a passive adversary which has obtained a physical access to a cryptographic device or its closeby environment. The goal of the adversary is to recover secret information within the device, assumed inaccessible from the outside, by monitoring physical variables issued from the device execution. These variables include the power consumption, the timing differences, the electromagnetic field emanations, or even the sounds that the device emits during the execution of a procedure.

In the scope of this thesis, two different kinds of side-channel attacks, both based on power analysis, are investigated. The first one is Simple Power Analysis (SPA) that tries to directly deduce information from measurements of the instantaneous power consumption. The second one is Differential Power Analysis (DPA) which makes a guess on a portion of secret information and tries to correlate it on different measurements of power consumption.

Measuring the power consumption of a device can be accomplished with various means. The most common one involves inserting a resistor in series with the power or ground input, and then digitally sampling the difference of voltage across the resistor at a certain rate using an oscilloscope [KJJ99]. Another technique, which will be used in this chapter, acquires the electromagnetic radiations that are generated by the current flows inside of the device circuitry [AJ01]. These emanations are directly related to the power consumption, and their acquisition requires less invasiveness, improves the localization of the power analysis from relevant parts of the circuitry, and enhances pattern matching [Hod+16]. This is done by placing a near-field magnetic probe on the processor chip at the location where the electromagnetic field is the highest. The collected samples are generally laid out on a graph as a function of time to create what is called a *power trace*. The precise location of the probe on our target device can be seen in Figure 4.1. An example of resulting power trace is illustrated on Figure 4.2.

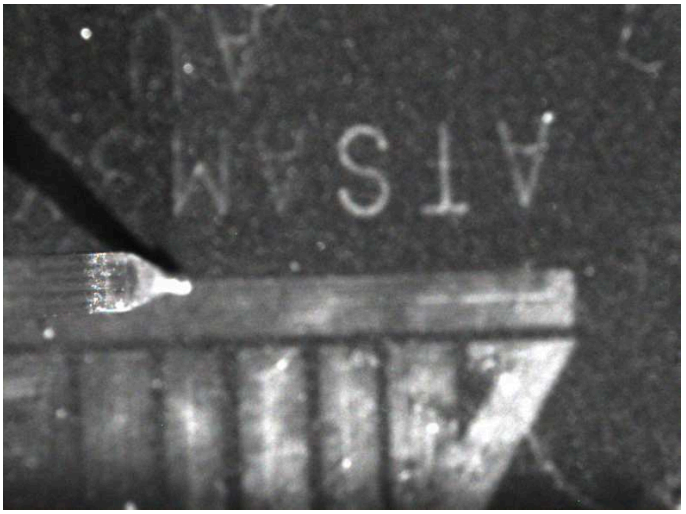


**Figure 4.1:** Position of the probe

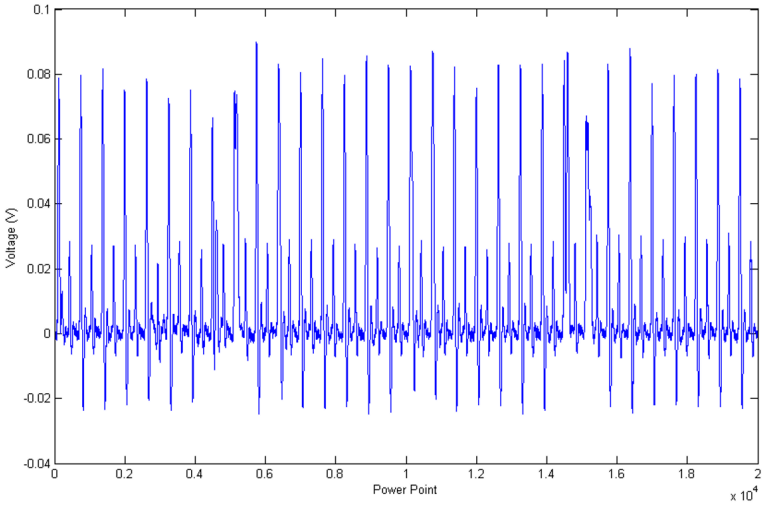



**Figure 4.2:** Example of a power trace [Hna+10]

The current chapter addresses the vulnerabilities related to power analysis of hash-based schemes presented in Chapter 3 using simple power analysis or differential power analysis. SPA is applied first on the whole signing procedure of our SPHINCS-256 implementation (see end of Section 3.4), then on the specific hash functions that handle secret values in SPHINCS-256. Based on the observations from the SPA, one of these hash functions is then specifically attacked using DPA.

## 4.1 Simple power analysis

Simple Power Analysis (SPA) is a typical side-channel attack on hardware that directly analyzes traces of power consumption from a device to recover information about its program. Since power consumption is known to depend on the processed operations

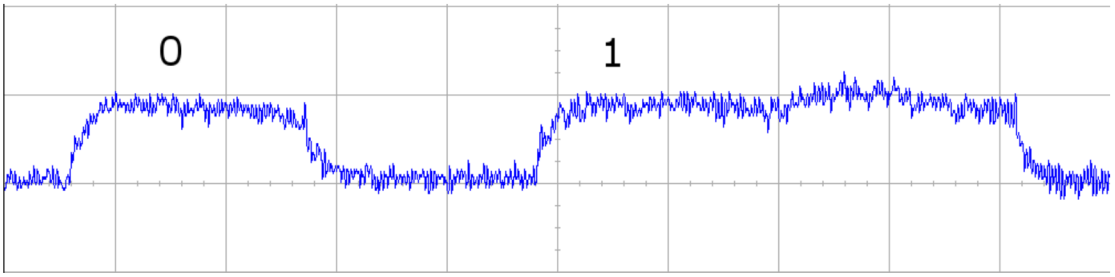



**Figure 4.3:** Power trace of the *square-and-multiply* technique in the context of RSA decryption. Each square represents $500\,\mu\text{s} \times 500\,\text{mV}$. On the traces, two operations occur: the first peak of energy corresponds to a *squaring*, which is shorter than the second peak of energy which corresponds to a *multiplication*. [Wik11]

[Koc+11] [BCO04], this analysis offers a good insight into cryptographic procedures running on hardware devices. The analysis consists of monitoring the device power consumption during the execution of cryptographic algorithms, and then cautiously examining the resulting traces to discern patterns.

A very famous example of a successful attack using SPA is the RSA key recovery from the power consumption of the decryption procedure which involves exponentiations with the *square-and-multiply* method [MDS99]. This technique consists of traversing the bits of the decryption key and performing a squaring or a multiplication operation depending on the value of the bits. Because the squaring operation requires less power than the multiplication one, the differences of consumption or timing between the two will be visible on a single power consumption trace. Making a distinction between those two allows therefore a complete reconstruction of the decryption key. An illustration of the previous analysis is shown on Figure 4.3.

In this section, the signing process of SPHINCS-256 will be attacked through simple power analysis. First, the procedure will be considered as a whole, in order to identify the macroscopic structure of the procedure in a single trace analysis, and determine whether information about secret keys can be deduced from it. Then, specific hash functions which manipulate secret values will be more cautiously analyzed in a multiple traces analysis.

### 4.1.1 Single trace analysis

Analyzing the full trace of SPHINCS signing process may already show data-dependent segments in the power consumption. This is especially the case with the presence of conditional branches or the use of CPU instructions with variable timing [Koc+11]. To investigate this, we made the device sign a dummy message with a known key and collected the resulting power trace.

Figure 4.4 shows an instance of a single full trace of the signing procedure of SPHINCS-256. In this trace, one can identify all the big steps of the signing process 3.4.2. The first small block corresponds to the digest computation and the index derivation (lines 1–6). Then, the next big block corresponds to the HOST signing procedure (line 9) which is followed by another big block corresponding to the HORST key generation (line 8). Finally, the rest of the trace represents the tree chaining computations (line 11–29).

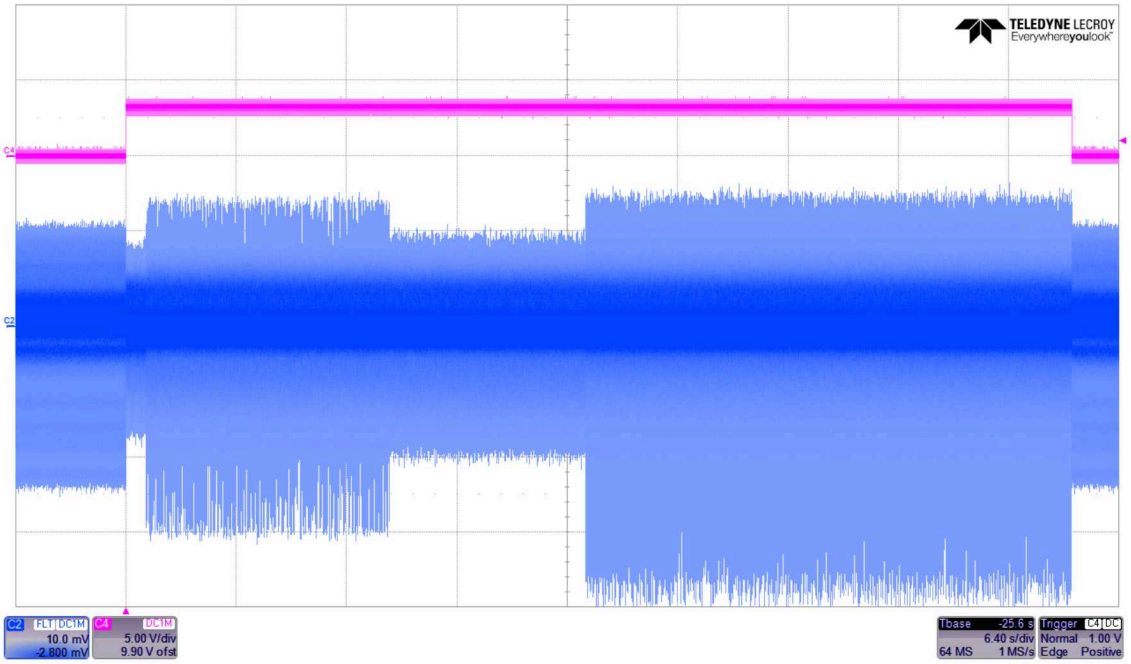



**Figure 4.4:** Full power trace of the SPHINCS signing procedure. The blue curve represents the voltage of SPHINCS signature when the magenta trigger is high.

By changing the message or the secret keys, the resulting traces are quite identical. The only observable data-dependence on the trace is the authentication path computations which are characterized by the space between the consumption spikes in the MSS or HORST signing procedures. The reason for these spikes comes from the discovery of new leaves, which involves calling the PRNG. However, this information is already part of the signature. This could however be interesting if a scheme relied on the secrecy of the authentication path.

Other than that, no secret information could be deduced from single trace analysis. This was to be expected, since the message and the secrets are handled with one of the latest hash functions that were designed to be as leakage-resilient as possible. Still, the trace characterizes well the execution of SPHINCS-256, which can be useful for identifying the scheme in a black-box context.

### 4.1.2 Multiple traces analysis

Even if the macroscopic consumption of SPHINCS did not show anything secret, it may be the case that microscopic power variations are visible during the procedures that handle secret values. The analysis therefore consists of isolating the execution of one function, in order to investigate the difference of power consumption between fixed inputs executions and random inputs executions.

In SPHINCS-256, four different functions process four different kinds of secrets (see Table 4.1). Being able to recover the input of any of these functions would permit an efficient attack on the scheme. Furthermore, as the output of PRNG $G_\lambda$ and the PRF $F_a$ are used to retrieve further secret values, they are of equal interest. Moreover, since some of these

| Function | | | Implementation |
|---|---|---|---|
| $(R_1, R_2)$ | $\leftarrow$ | $F(M, \text{SK}_2)$ | BLAKE-512 |
| $\text{SEED}_A$ | $\leftarrow$ | $F_a(A, \text{SK}_1)$ | BLAKE-256 |
| $X_A$ | $\leftarrow$ | $G_\ell(\text{SEED}_A)$ | ChaCha12 |
| $Y_A$ | $\leftarrow$ | $c^{W-1}(X_A)$ | ChaCha12 permutation |

**Table 4.1:** Summary of the functions implementations in SPHINCS-256.

outputs are directly linked to the inputs of other functions, combining the information to perform "Meet-In-The-Middle" attacks [KH10] may present a potential risk.

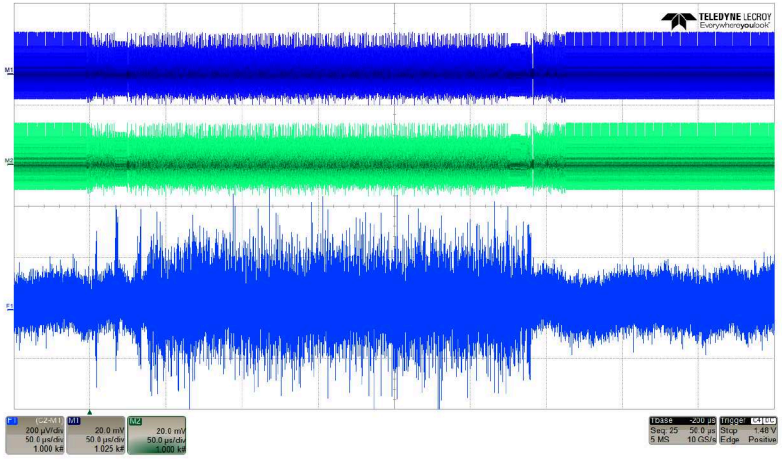
Figure 4.5 shows the multiple traces analyses for the four functions from Table 4.1. Each subfigure consists of three plots: the upper one shows the average of a thousand traces when the functions are executed with fixed inputs, the one in the middle shows the average of a thousand traces when the functions are executed with random inputs, and the lower one shows the difference between the two averages. A spike in the last curve shows the dependence of the power consumption on the inputs. The analysis aims to exploit the pattern of spikes.

As one can observe, many spikes appear on the graphs, which confirms the presence of a relation between the power consumption and the inputs of the function. For some functions, the spikes at the beginning of a few graphs, such as the PRNG, correspond to the input being read by the device and should be discarded in the analysis. Even if there is a relation between the inputs and power, the distinction of simple patterns is still unclear. This means that simple power analysis fails to deduce any secret information from the power traces of hash functions. One could zoom again to analyze the specific instructions in order to push the simple power analysis further. Still, the existence of the leakage at this point of the analysis is sufficient to conduct differential power analysis.

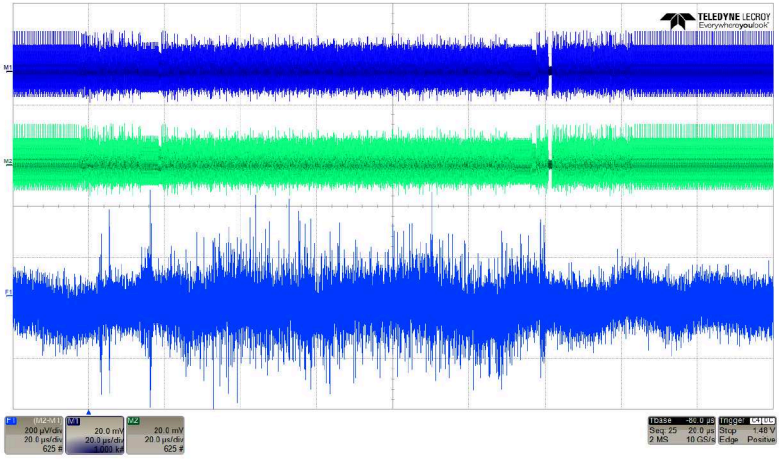
## 4.2 Differential power analysis

Differential Power Analysis (DPA) is a more advanced hardware side-channel attack which exploits the statistical bias from the power traces to confirm hypothetical guesses made on secret values. The strategy of such an attack first collects many power traces of one specific operation for different inputs, then tries to find a correlation between the instantaneous consumption and bits from intermediate values that depend on secret data. The process is typically run on small chunks of secret values and repeated through all the chunks, so the key recovery complexity is reduced.

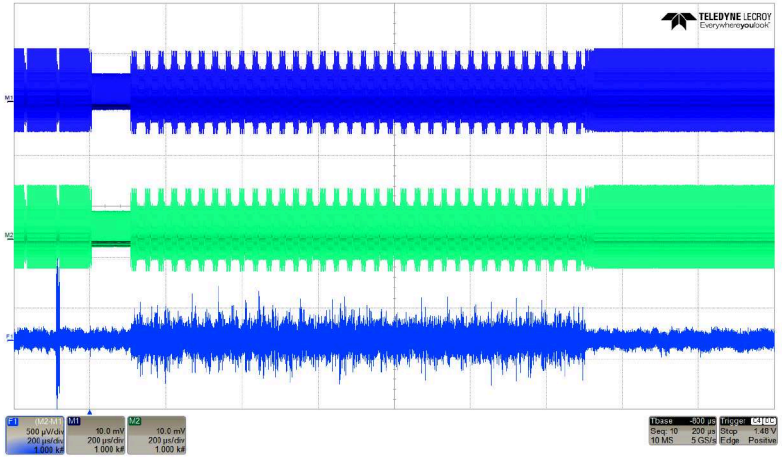
The original DPA attack from Paul Kocher et al. is based on the fact that, at a precise point in the power trace, the frequency of the instantaneous power consumption will follow a certain distribution depending on the value of a few bits from an intermediate quantity [Koc+11]. Consider the following scenario: assume that an attacker wants to recover the key from secure cipher, such as AES, implemented in hardware. Suppose that the device can be used to encrypt many known plaintexts, and that the power consumption of each encryption can be monitored. Furthermore, let us pretend that the attacker identifies



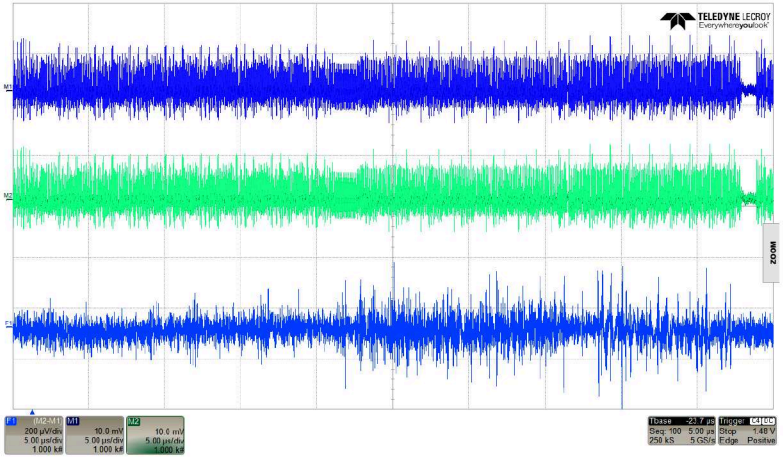
(a) BLAKE-512



(b) BLAKE-256



(c) ChaCha12



(d) ChaCha12 permutation

**Figure 4.5:** Multiple traces analyses for specific hash functions involved in SPHINCS-256.

the precise point in the power trace at which the process of an intermediate value of the encryption occurs. Now, if the attacker tries to compute this intermediate value from a plaintext by making a guess on the key, the power consumption samples at this point should follow the distribution that corresponds to the computed intermediate value. In other words, the voltage samples for all the power traces should all be correlated to the computed intermediate value corresponding to the key guess. The goal of the attacker is therefore to measure this correlation, and choose the key guess for which the difference of distributions is the more apparent.

There are many ways of measuring the correlation between a sample distribution and an intermediate value. The original one, proposed by Paul Kocher et al., computes the *difference of means* which consists of assigning a single bit of the presumed intermediate value to all the voltage samples at a precise point in the power trace. Then, the method splits these distributions in two partitions depending on the value of the bit, and computes the difference of empirical mean between the sample distributions of these two partitions. Big differences demonstrate good correlation between the sample and the value.

One way to improve the method is to make an additional assumption on the source of the power consumption. The most common one is the supposition that the voltage of a power consumption spike is proportional to either the Hamming weight of the data being processed or the Hamming distance between the change of data [BCO04]. These models allow more elaborated correspondence measurements, such as the *variance test* (V-test), investigated by François-Xavier Standaert, which considers the variance of the partitions that depend on the Hamming weight of the intermediate values [SGV08]. Moreover, the *Pearson correlation*, studied by Eric Brier, which computes the correlation factor between the Hamming weight distance and the voltage samples can also be considered [BCO04].

The goal of section is to apply these correlation methods to prove that there is indeed a relation between portions of secret values and the power consumption of a specific hash function, as a follow-up of the study from SPA. Since the BLAKE-256 hash function is called several times with the first SPHINCS secret key but different addresses, this makes a perfect target for a DPA attack. Once the relation has been established for BLAKE-256, the correspondence can be exploited by an attacker to recover bits from the first SPHINCS secret key.

### 4.2.1 Leakage proof on BLAKE-256

In SPHINCS-256, the hash function BLAKE-256 uses the first secret key and a known address to derive a secret seed for the key generation of W-OTS$^+$ instances. In order to confirm guesses made on this secret key with differential power analysis, a correspondence between collected power traces and an intermediate value first needs to be established. This is attempted by collecting multiple power traces for different addresses, computing the value of a certain intermediate quantity with the addresses, and applying different correlation measurements between the two.

To choose a good intermediate value for DPA, a cautious examination of the BLAKE-256 compression algorithm needs to be conducted. In the context of SPHINCS-256, the function input consists of 16 values $m_i$ of 32 bits, each of which are shown in Table 4.2. The first steps of the main loop in BLAKE-256 compression function are roughly summarized

in Algorithm 4.2.1. These steps involve a mixing subroutine `Mix` shown in Algorithm 4.2.2 which mixes four intermediate values with two chunks of the functions input.

**input :** $(m_0, \ldots, m_{15})$ – BLAKE-256 input (c.f. Table 4.2)

---

1 $v_0 \leftarrow$ `0x6A09E667`, $v_4 \leftarrow$ `0x510E527F`, $v_8 \leftarrow$ `0x243F6A88`, $v_{12} \leftarrow$ `0xA4093822`
2 $v_1 \leftarrow$ `0xBB67AE85`, $v_5 \leftarrow$ `0x9B05688C`, $v_9 \leftarrow$ `0x85A308D3`, $v_{13} \leftarrow$ `0x299F31D0`
3 $v_2 \leftarrow$ `0x3C6EF372`, $v_6 \leftarrow$ `0x1F83D9AB`, $v_{10} \leftarrow$ `0x13198A2E`, $v_{14} \leftarrow$ `0x082EFA98`
4 $v_3 \leftarrow$ `0xA54FF53A`, $v_7 \leftarrow$ `0x5BE0CD19`, $v_{11} \leftarrow$ `0x03707344`, $v_{15} \leftarrow$ `0xEC4E6C89`

5 $v_{12} \leftarrow v_{12} \oplus m_{15}$
6 $v_{13} \leftarrow v_{13} \oplus m_{15}$

7 `Mix`$(v_0, v_4, v_8, v_{12}; m_0)$
8 `Mix`$(v_1, v_5, v_9, v_{13}; m_2)$
9 `Mix`$(v_2, v_6, v_{10}, v_{14}; m_4)$
10 `Mix`$(v_3, v_7, v_{11}, v_{15}; m_6)$

11 `Mix`$(v_3, v_4, v_9, v_{14}; m_{14})$
12 `Mix`$(v_2, v_7, v_8, v_{13}; m_{12})$
13 `Mix`$(v_0, v_5, v_{10}, v_{15}; m_8)$
14 `Mix`$(v_1, v_6, v_{11}, v_{12}; m_{10})$

**Algorithm 4.2.1:** Rough summary of the first round from the BLAKE-256 compression function.

**input :** $(v_a, v_b, v_c, v_d)$ – Intermediate values
**input :** $m_e$ – Chunk of message (c.f. Table 4.2)

---

1 $c_0 \leftarrow$ `0x243F6A88`, $c_4 \leftarrow$ `0xA4093822`, $c_8 \leftarrow$ `0x452821E6`, $c_{12} \leftarrow$ `0xC0AC29B7`
2 $c_1 \leftarrow$ `0x85A308D3`, $c_5 \leftarrow$ `0x299F31D0`, $c_9 \leftarrow$ `0x38D01377`, $c_{13} \leftarrow$ `0xC97C50DD`
3 $c_2 \leftarrow$ `0x13198A2E`, $c_6 \leftarrow$ `0x082EFA98`, $c_{10} \leftarrow$ `0xBE5466CF`, $c_{14} \leftarrow$ `0x3F84D5B5`
4 $c_3 \leftarrow$ `0x03707344`, $c_7 \leftarrow$ `0xEC4E6C89`, $c_{11} \leftarrow$ `0x34E90C6C`, $c_{15} \leftarrow$ `0xB5470917`

5 $v_a \leftarrow v_a + (m_e \oplus c_{e+1}) + v_b$
6 $v_d \leftarrow (v_d \oplus v_a) \lll 16$
7 $v_c \leftarrow v_c + v_d$
8 $v_b \leftarrow (v_b \oplus v_c) \lll 12$

9 $v_a \leftarrow v_a + (m_{e+1} \oplus c_e) + v_b$
10 $v_d \leftarrow (v_d \oplus v_a) \lll 8$
11 $v_c \leftarrow v_c + v_d$
12 $v_b \leftarrow (v_b \oplus v_c) \lll 7$

**Algorithm 4.2.2:** Mixing function of the BLAKE-256 compression function.

By analyzing the propagation of the inputs in the mixing function, one can observe that after the first half of mixing (line 11), all the $v_i$ for $0 \leq i < 16$ are mixed with the whole

| Input | Value | Input | Value |
|---|---|---|---|
| $m_0$ | $\text{SK}_1[0{:}32]$ | $m_8$ | $A[0{:}32]$ |
| $m_1$ | $\text{SK}_1[32{:}64]$ | $m_9$ | $A[32{:}64]$ |
| $m_2$ | $\text{SK}_1[64{:}96]$ | $m_{10}$ | `0x80000000` |
| $m_3$ | $\text{SK}_1[96{:}128]$ | $m_{11}$ | `0x00000000` |
| $m_4$ | $\text{SK}_1[128{:}160]$ | $m_{12}$ | `0x00000000` |
| $m_5$ | $\text{SK}_1[160{:}192]$ | $m_{13}$ | `0x00000001` |
| $m_6$ | $\text{SK}_1[192{:}224]$ | $m_{14}$ | `0x00000000` |
| $m_7$ | $\text{SK}_1[224{:}256]$ | $m_{15}$ | `0x00000140` |

**Table 4.2:** Summary of BLAKE-256 inputs.

secret key $\text{SK}_1$. Let $s_i$ denote the 32-bit chunks of $\text{SK}_1$ for $0 \leq i < 8$. Table 4.3 quickly shows which values depend on which chunks of the secret key at this point. A particularly interesting call for DPA is the third next call (line 13) which mixes $v_0$, $v_5$, $v_{10}$, and $v_{15}$ with $m_8$ and $m_9$, as it involves values mixed with the secret key and the address value that is provided in the signature. More precisely, if the mixing function `Mix` is unrolled, we see that the operation $v_0 = v_0 + (m_8 \oplus c_9) + v_5$ involves only $v_0$ and $v_5$, which respectively depend on $s_0$, $s_1$, and $s_2$, $s_3$. Therefore, the value of $v_0$ at this point will be taken as the intermediate value for DPA.

$$\begin{array}{rcl} (v_0,\, v_4,\, v_8,\, v_{12}) & \propto & (s_0,\, s_1) \\ (v_1,\, v_5,\, v_9,\, v_{13}) & \propto & (s_2,\, s_3) \\ (v_2,\, v_6,\, v_{10},\, v_{14}) & \propto & (s_4,\, s_5) \\ (v_3,\, v_7,\, v_{11},\, v_{15}) & \propto & (s_6,\, s_7) \end{array}$$

**Table 4.3:** Summary of the secret key dependence for intermediate values.

Now that the intermediate value is identified, we isolated the BLAKE-256 function in our device to easily collect power traces. The secret key was fixed, and the function was called $q = 1000$ times with different addresses for $A$ each time while monitoring the power consumption of the device. In theory, as the hash function is called $32 \times 12 = 384$ times during a full SPHINCS signature, 5 different signatures should give at least a thousand different power traces. However, in our case, only the first half of the address is involved in the intermediate value, and since this part is the same among the 32 calls, only 12 traces can be collected per signature. Since the top subtrees are likely to repeat, this means that approximately 250 signatures to obtain a thousand different traces[1].

Assume we collected a thousand power traces $T_i[j]$ of $N$ samples for $0 \leq j < N$ corresponding to a thousand different addresses $A_i$ and that we know $\text{SK}_1$. The intermediate value $v_0^i$ can therefore be exactly computed for $0 \leq i < q$. The following will try several measurements of correspondences, such as *difference of means*, *V-test*, and *Pearson correlation* to prove the relation between $T_i$ and $v_0^i$.

[1]The actual expected number of different traces is $\sum_{i=0}^{d-1} 2^i \left(1 - ((2^i - 1)/2^i)^q\right)$

### Difference of means

With difference of means, a single bit of the intermediate values is targeted. The traces are then divided in two partitions according to the value of this bit. The method consists of traversing all the points in the traces, take all the samples in one partition at this point, and compute their mean. The difference between the means of the two partitions at each point is then computed. Large differences between the two means at some point in the power trace characterizes the dependence between the intermediate value bit and the power consumption.

Let $D(v_0^i, b) \in \{0, 1\}$ be the value of the targeted bit $b$ in $v_0$. The difference of means $\Delta_D[j]$ for all samples $0 \leq j < N$ is defined by the the following quantity [KJJ99]:

$$\Delta_D[j] = \frac{\sum_{i=0}^{q-1} D(v_0^i, b)T_i[j]}{\sum_{i=0}^{q-1} D(v_0^i, b)} - \frac{\sum_{i=0}^{q-1}(1 - D(v_0^i, b))T_i[j]}{\sum_{i=0}^{q-1}(1 - D(v_0^i, b))}.$$

Figure 4.6 shows the results of computing $\Delta_D[j]$ for the isolated computation of $v_0^i$ inside of the BLAKE-256 compression function. In order to differentiate the computation of $v_0$ from the rest, we explicitly added a delay of $1\,\mu$s between the operations. The upper plot represents an average of all the power traces $T_i[j]$, while the one below represents the difference of means $\Delta_D[j]$ for four different scenarios. The plot in blue is $\Delta_D[j]$ when $v_0^i$ was computed with the correct key, while the three other plots when the key chunks for $s_0$, $s_1$, $s_2$, and $s_3$, differ in the last 4 bits. As one can observe, the spikes in the plot are the greatest on the blue curve, which indicates that there is indeed a relation between $T_i[j]$ and $v_0^i$ for some $j$.

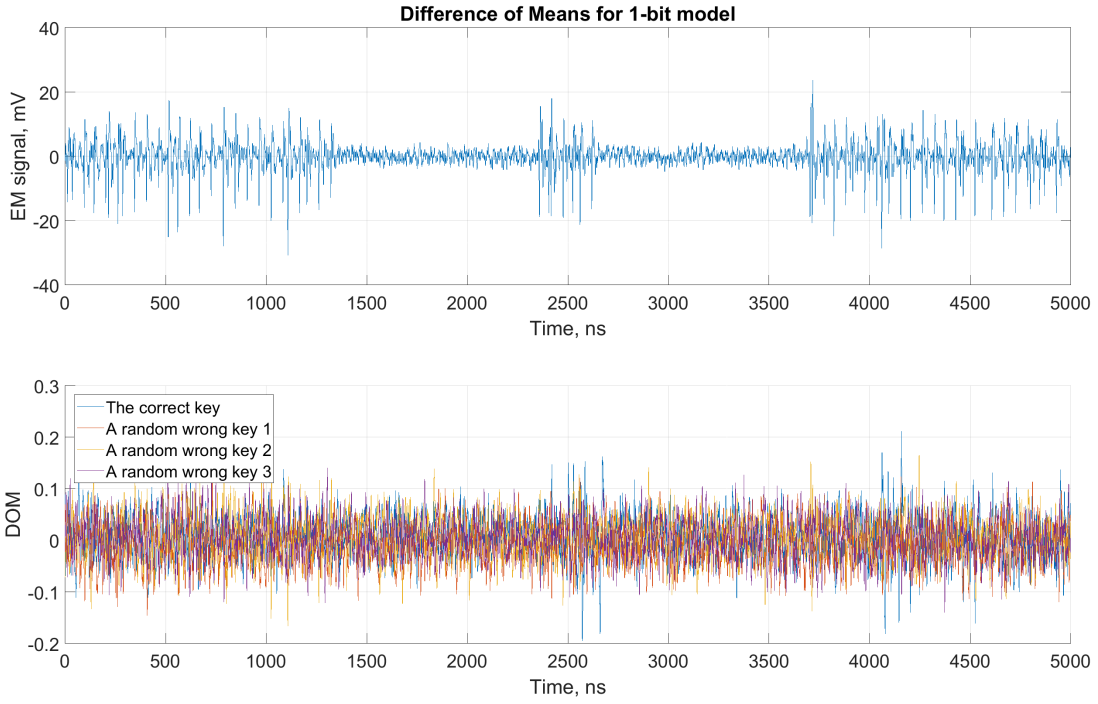



**Figure 4.6:** Results for the difference of means during $v_0$ computation ($q = 1000$ traces)

**V-test**

A variance test partitions the intermediate values in terms of Hamming weight to compute a ratio of variances. First, for a fixed point in the trace, the sample variance in all the traces is computed. Then, the variances of the sample from the partitions weighted with the size of the partition is averaged. The ratio between the two is maximized when the intermediate values are in direct relation with the samples.

Let $\sigma^2[j]$ be the empirical variance of the sample $0 \leq j < N$ in all the traces $T_i$. Compute the variance of the same sample for each partition $k$. Attribute the different $T_i$ to the Hamming weight of the intermediate values $v_0^i$, resulting in 32 partitions $P$. Let $N_k$ be the size of each partition $0 \leq k < 32$ and compute the variance of the same sample in each partition $\sigma_k^2$. The V-test distinguisher is then given by the following ratio [Gie+09]:

$$\sigma_P^2[j] = \frac{\sigma^2[j]}{\frac{1}{N}\sum_{k=0}^{31} N_k \sigma_k^2}.$$

Figure 4.7 shows the results of computing $\sigma_{k'}^2[j]$ for the isolated computation of $v_0^i$ inside of the BLAKE-256 compression function. As explained before, we explicitly added a delay of $1\mu$s between the $v_0^i$ computation for the purpose of the study. The upper plot represents an average of all the power traces $T_i[j]$, while the one below represents the V-test ratio $\sigma_{k'}^2[j]$ on 4 bits of $v_0^i$ for four different scenarios. The plot in blue is $\sigma_{k'}^2[j]$ when $v_0^i$ was computed with the correct key, while the three other plots when the key chunks for $s_0$, $s_1$, $s_2$, and $s_3$, differ in the last 4 bits. As one can observe, the spikes in the plot are the greatest on the blue curve, which indicates that there is indeed a relation between $T_i[j]$ and $v_0^i$ for some $j$.

**Pearson correlation**

The Pearson correlation coefficient evaluates the linear fit between the Hamming distance of the intermediate value from some initial value and a power samples distribution. The expression finds the experimental covariance between the two variables and normalizes it with the product of their experimental variance. A linear correlation is indicated by a coefficient close to 1 or $-1$.

Let $H_{i,R} = H(v_0^i \oplus R)$ be the Hamming distance between the intermediate values $v_0^i$ and the initial value $R$, supposed to be unknown but the same through all the executions. The Pearson correlation coefficient is therefore [BCO04]:

$$\rho_R[j] = \frac{N\sum_{i=0}^{q-1} T_i[j]H_{i,R} - \sum_{i=0}^{q-1} T_i[j]\sum_{i=0}^{q-1} H_{i,R}}{\sqrt{N\sum_{i=0}^{q-1} T_i[j]^2 - \left(\sum_{i=0}^{q-1} T_i[j]\right)^2}\sqrt{N\sum_{i=0}^{q-1} H_{i,R}^2 - \left(\sum_{i=0}^{q-1} H_{i,R}[j]\right)^2}}.$$

Figure 4.8 shows the results of computing $\rho_R[j]$ for the isolated computation of $v_0^i$ inside of the BLAKE-256 compression function. Once again, we explicitly added a delay of $1\,\mu$s between the $v_0^i$ computation to differentiate it from the rest. In our case, we considered the Hamming weight model, so we used $R = 0$. The upper plot represents an average of all the power traces $T_i[j]$, while the one below represents the correlation factor $\rho_R[j]$ on 4 bits of $v_0^i$ for four different scenarios. The plot in blue is $\rho_R[j]$ when $v_0^i$ was computed with the correct key, while the three other plots when the key chunks for $s_0$, $s_1$, $s_2$, and

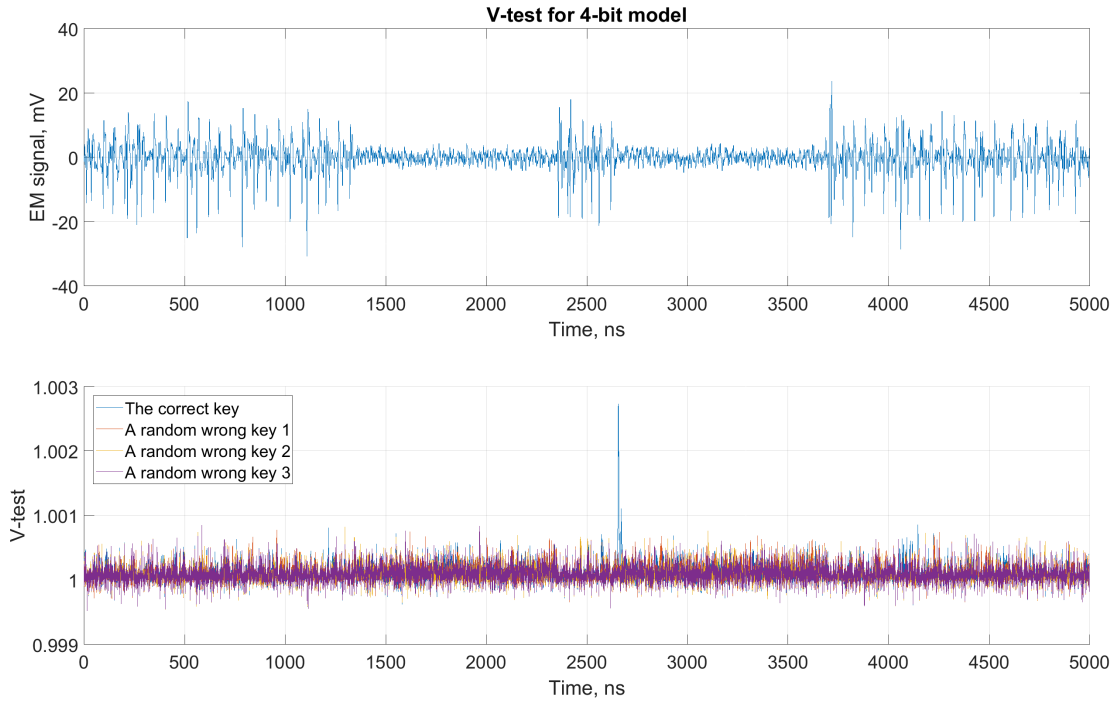



**Figure 4.7:** Results for the V-test on 4 bits of $v_0$ during its computation ($q = 1000$ traces)

$s_3$, differ in the last 4 bits. As one can observe, the spikes in the plot are the greatest on the blue curve, which again shows that there is indeed a relation between $T_i[j]$ and $v_0^i$ for some $j$.

### Countermeasures

Since a device cannot detect that its power consumption is being monitored, preventing a differential power analysis demands an additional effort in the implementation of the cryptosystem. The countermeasures deployed aim to either limit the collection of power traces or their dependence on data. This can be achieved physically, which involves shielding the device to directly prevent the measurement, or algorithmically by modifying the procedures.

The two most common algorithmic ways of protecting against power analyses are *hiding* and *masking*. The former consists of drowning the computation of the values amongst the others such that the power consumption becomes unintelligible or different at each execution, while the latter consists of operating on biased data in such a way that the intended result can then be easily unbiased. A way of hiding the power consumption of BLAKE-256 is to randomize the time at which an operation is performed, so the operations in the produced power traces are unlikely to occur at the same time which requires the attacker to re-align the power traces. Masking, however, does not really apply in the context of protecting hash functions, as there is usually no way to recover an intended output from the hash value of a masked input.

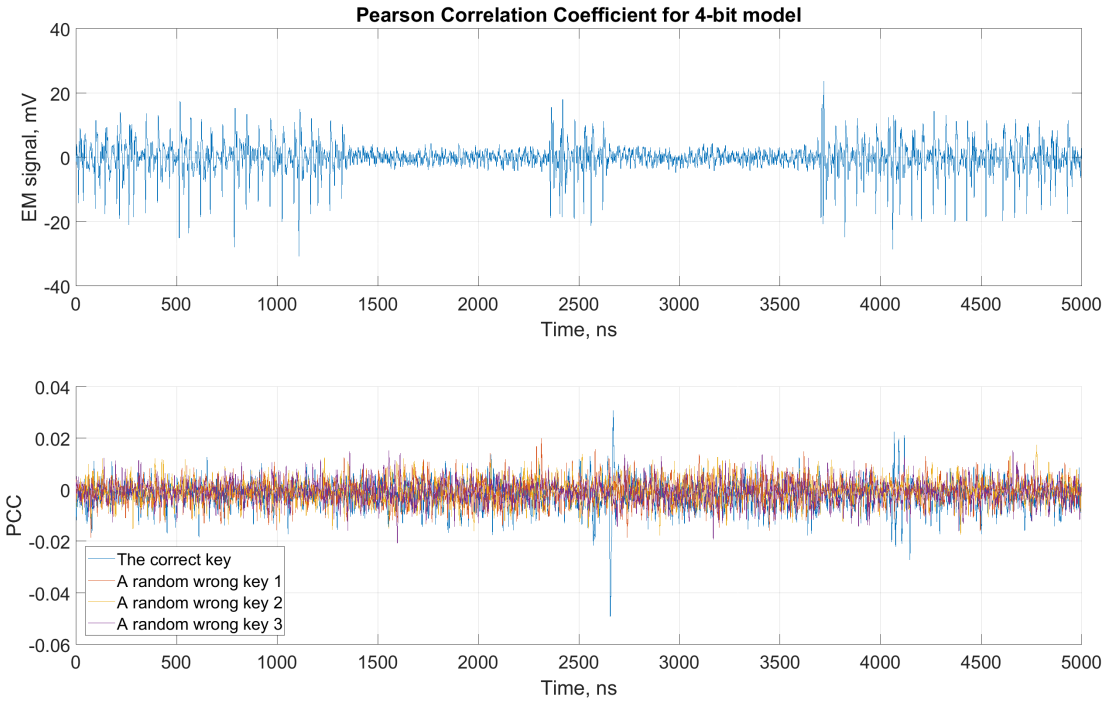



**Figure 4.8:** Results for the Pearson correlation on 4 bits of $v_0$ during its computation ($q = 1000$ traces).

The particular change that could be considered in the design of SPHINCS-256 is to first limit the use of the BLAKE-256 function. This could be done by calling the pseudorandom function only once at each subtree and using the resulting seed for the generation of all the W-OTS$^+$ secret keys. Ideally, hiding the address in such a way that the value used is secret to the attacker would be preferable. This could be done by computing the hash of the address using the second SPHINCS secret key $\text{SK}_2$, or even a third secret key. This would add a second step of DPA before being able to break the scheme, which could be considerably harder, since the attacker has no way of identifying whether the recovery was successful, as opposed to recovering $\text{SK}_1$.

# Chapter 5

# Fault Attacks

A fault is a hardware or software failure that unexpectedly causes the malfunction of a device due to an anomalous execution or deterioration. Multiple types of faults exist; for instance, a fault is *latent* if the error is due to the limitations of hardware, or *transient* when the result of an operation is accidentally corrupted, typically because of ephemeral alterations within the device, known as *glitches*. The misbehavior caused by transient faults during cryptographic procedures may be exploited to deduce secret information, since such information may be related to the erroneous result. Consequently, evaluating the security of cryptosystems in the presence of transient faults is important, as they can be intentionally induced by a malicious attacker, leading to a Fault Attack (FA).

Inducing transient faults to attack a cryptosystem results in a model in which an active adversary has acquired physical access to a cryptographic device containing supposedly inaccessible secrets, such as decryption keys. The goal of the adversary is to break the cryptographic schemes by manipulating the device at will, but with an additional assumption: any kind of hardware faults can be injected. These faults are characterized by their *granularity*, i.e, the number of bits that requires to be affected, the bit *modification* that the adversary needs to perform, the required *control* of the bit location and the time to inject the fault, and the *target* value or procedure [Bal14]. These characteristics generally depend on the power and skills of the adversary.

The techniques that deliberatedly inject hardware faults are essentially characterized depending on their cost, the skills required, their degree of invasiveness, and the damage caused. On one hand, low-cost techniques usually require low to moderate skills, are non to semi-invasive, and typically produce random errors during a procedure that leave little to no damage. They include underfeeding the device, injecting a glitch in the power supply, altering the clock signal, overheating the dynamic random-access memory (DRAM), producing strong electromagnetic disturbances close to the device, and exposing the circuit to a light source. High-cost ones, on the other hand, generally require high skills, are semi to fully invasive, and can produce precise alterations of the procedures that may permanently damage the device. They involve focusing a precise light beam, laser beam, or even ion beam, on a small part of the circuit.

This chapter investigates the vulnerabilities of the hash-based digital signature schemes introduced in Chapter 3 related to fault attacks. First, an analysis of non- to semi-invasive attacks against MSS with PRNG, OTS and FTS, and CMSS is conducted. Then, invasive attacks which involve a much more powerful adversary will be considered against W-OTS.

## 5.1 Non-invasive attacks

This section covers non-invasive and semi-invasive attacks that usually require low-cost equipment. The admitted model of faults that an attacker can introduce can affect one or many bits from a single register by either flipping or randomizing them, while not permanently damaging the device. Such a model is typically achieved with voltage glitching.

### 5.1.1 Disabling MSS with PRNG

Using a pseudorandom number generator to improve a stateful Merkle scheme—as explained in Section 3.4.1—makes the resulting system vulnerable to hardware or software errors. Because the occurrence of errors during the pseudorandom number generation causes the scheme to end up with a wrong seed, the whole cryptosystem cannot be used anymore. Thus, a whole instance can be completely disabled with the injection of a single fault.

This attack targets any variant of a stateful Merkle scheme using a pseudorandom number generator (e.g., XMSS). Assume the current seed of the scheme is $\text{SEED}_{\text{MSS}_i}$ and that a fault happens on the output of the next seed $\text{SEED}'_{\text{MSS}_{i+1}}$ such that it deviates from its expected output. This corrupted seed will therefore generate another wrong seed for its corresponding one-time signature $\text{SEED}'_{\text{OTS}} \neq \text{SEED}_{\text{OTS}}$ that will derive wrong secret values $\hat{X}_{i+1,\,j} \neq X_{i+1,\,j}$ for $0 \leq j \leq m$. Additionally, the next seeds responsible for the upcoming instances of OTS will also differ from expected, making the scheme disfunctional. The process is illustrated in Figure 5.1 which shows how a corrupted output propagates all the way to the rest of the pseudorandom key generation.

The purpose of the pseudorandom number generator was to recover all the one-time signature secrets in a more convenient way. However, since all the values have to be discovered at least once to generate the overall public-key, the signer must rely on the reproductibility of the number generation. Because the fault causes the scheme to deviate from the expected pseudorandom generation, the secret values are no longer correctly selected, making the signature necessarily invalid. Moreover, since only the seed acts as the overall state of the scheme and because it is updated every time the device signs a new message, there is no way to recover from it.

**Fault characteristics**

- **Number of faulty signatures** : 1
- **Granularity** : a single bit to many bits
- **Modification** : random
- **Control** : loose
- **Target** : any value in the first pseudorandom number generation

Such an attack makes the device running the pseudorandom number generator improvement doomed to produce invalid signatures for the rest of its lifetime. Therefore, its implementation is hardly practical, especially for embedded devices where this model of

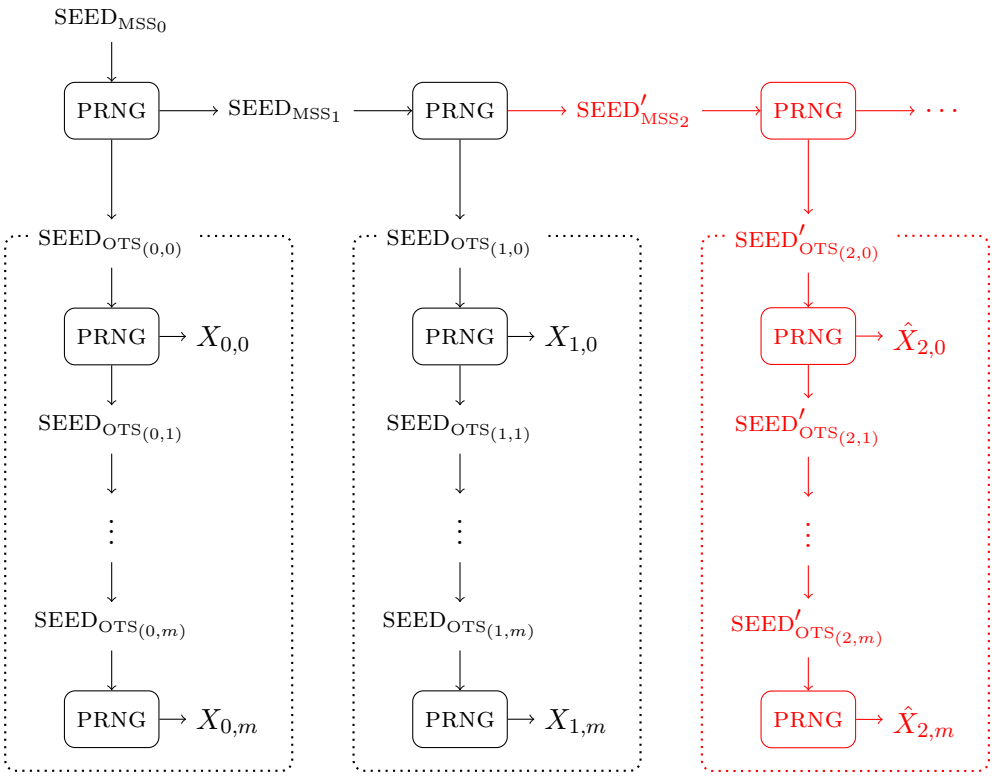



**Figure 5.1:** Faulting the OTS secret key generation using a PRNG in the context of MSS.

fault is completely applicable. More importantly, the scheme can break itself because of accidental transient errors.

**Countermeasures**

There is no best way to protect against this attack, unless the signer computes every single seed in advance and scraps them properly after usage. In this case, the fault will only cause one signature to be wrong. To trade-off with memory, the signer can proceed by levels and compute one in $m$ seeds, with an arbitrary $0 < m < 2^H$, so it can recover from the failure.

The scheme can also compute many times the resulting seed of the first PRNG and compare the results in order to detect an error. The key generation process would only proceed if all the comparisons succeeded. However, since fault injection also allows to skip instructions, this countermeasure requires finer targeted faults.

Another way of permitting a recovery is to keep an emergency backup of the first seed. In this case, the signing scheme can go back to the correct state if a wrong signature is detected. The obvious problem with this solution is that perfect secrecy is no more provided.

Finally, note that since the model allows the attacker to sign as many messages as wanted, all the seeds can simply be depleted by forcing the device to sign all the available messages.

### 5.1.2 Existential forgeries on OTS and FTS

In the scenario in which an OTS or FTS instance is used multiple times to sign the same arbitrary message (as it is the case in the tree chaining method, for instance), the component becomes vulnerable to the presence of faults during its signing process. A fault during the digest computation makes the component reveal incorrect secret values. An attacker can therefore exploit this vulnerability to collect many secret values until another message can be forged.

Suppose there is a device running an authentication scheme which uses a same instance of OTS or FTS to always sign the same message (e.g., SPHINCS, XMSS). Let $M \in \{0,1\}^*$ be the message, $D \in \{0,1\}^n$ its corresponding digest of $n$ bits, and $\sigma$ the resulting signature. Assume the same message is being signed $q$ more times, but an error occurs during each of the digest computations, making the resulting hash value random. This causes the device to produce $q$ faulty signatures $(\hat{\sigma}_1, \ldots, \hat{\sigma}_q)$ which all contain secret values different from the valid signature. By using the valid signature, the faulty digests $(\hat{D}_1, \ldots, \hat{D}_q)$ can be recovered, so the secret values from the faulty signatures can be identified. Finally, by combining the secret values learnt in $\sigma$ and $(\hat{\sigma}_1, \ldots, \hat{\sigma}_q)$, an attacker may potentially forge a valid signature $\sigma'$ corresponding to a different $M' \neq M$ issued from the same instance. The probability of success of this forgery increases with the amount of different secret values learnt, hence with the number of faulty signatures $q$ that an attacker can collect.

**Fault characteristics**

- **Number of faulty signatures** : $q$
- **Granularity** : a single bit flip to many bits
- **Modification** : random
- **Control** : loose
- **Target** : any value in the message digest computation

**Countermeasures**

In the case the memory allows it, a same signature can be computed once and output every time needed in order to limit its reuse. A fault can therefore never compromise this component security, as the signature is not processed anymore. For heavy schemes, such as CMSS, caching the frequent OTS can be considered. This would force an attacker to get around the caching which can make the attack considerably harder.

Another way to thwart the attack is to compute the message digest many times and compare the results. The signing procedure would only sign if all the digests are the same, and resets or safely fails otherwise. This is affordable, since the digest computation should be fast relative to the rest of the signing procedure. The number of comparisons should be chosen according to the risk of fault attacks, as the attacker can still skip the comparison instructions using non-invasive faults as well.

The following subsections address the success probability of the attack specific to OTS and FTS schemes and therefore extends the study from Leon G. Bruinderink and Andreas Hülsing who studied the security of OTS under two-message attacks [BH16].

#### 5.1.2.1 LD-OTS

Assume the previously described attack occurs on a Lamport–Diffie one-time signature component, as explained in Section 3.1.1. The faulty signatures will contain secret values corresponding to the binary representation of the faulty message digest. The goal of the adversary is to find another message such that the bits of its digest are all covered by the bit union of all the faulty digests.

After collecting the faulty signatures, the attack recovers the faulty digests by comparing the values inside of the faulty signatures with the ones from $\sigma$. Let $(d_0^j, \ldots, d_{n-1}^j)_2$ be the binary representation of the valid digest for $j = 0$ and the faulty ones for $0 < j \leq q$. The forgery success for a message $M' \neq M$ and its equivalent digest $D' = (d'_0, \ldots, d'_{n-1})_2$ is then defined by the following event:

$$\text{Success} = \forall\ 0 \leq i < n\ \exists\ 0 \leq j \leq q\ \ d_i^j = d'_i.$$

Since every bit is uniformly random, the probability that, at index $i$, at least two $d_i^j$ have a different bit value is $(1 - 1/2^q)$. If this occurs, then the value for $d'_i$ is necessarily known. Otherwise, all the digests have the same bit at index $i$, but there is still a chance of $1/2$ that $d'_i$ also shares this value. By putting these results together, we have:

$$\mathbb{P}\left(\exists\ 0 \leq j \leq q\ \ d'_i = d_i^j\right) = \left(1 - \frac{1}{2^q}\right) \cdot 1 + \left(\frac{1}{2^q}\right) \cdot \frac{1}{2}.$$

Since all bits are drawn uniformly, the forgery success probability becomes:

$$\mathbb{P}(\text{Success}) = \left(1 - \frac{1}{2^{q+1}}\right)^n.$$

#### 5.1.2.2 W-OTS($^+$)

Suppose now that the component attacked runs any variant of the Winternitz one-time signature scheme, as described in Section 3.1.2 or Section 3.4.5. In this case, the faulty signatures contain a point in the hash chain that corresponds to the value of bits blocks in the digest. The adversary needs to find another message such that all the blocks in the digest have equal or higher values than the lowest ones in the collected signatures.

To recover the values of the faulty digests, the attacker can first derive the public key $Y_{\text{W-OTS}}$ from $\sigma$. Then, the values inside of the faulty signatures can be hashed several times, until the values from the public key are reached[1]. Let $(b_0^j, \ldots, b_{\ell_1-1}^j)$ be the result of splitting the valid digest for $j = 0$ and the faulty ones for $0 < j \leq q$ into blocks of length $w$. Moreover, let $C = \sum_{i=0}^{\ell_1-1}(W - b_i^j)$ be also split in $(b_{\ell_1}^j, \ldots, b_{\ell-1}^j)$ for the same $0 \leq j \leq q$. A message $M' \neq M$ and its equivalent digest $D' = (b'_0, \ldots, b'_{\ell_1-1})$ is then successfully forged if the following event occurs:

$$\text{Success} = \forall\ 0 \leq i < \ell\ \exists\ 0 \leq j \leq q\ \ b_i^j \leq b'_i.$$

[1] In the case of W-OTS$^+$, the values of the masks can be brute-forced until the correct public-key value is reached.

To derive the success probability for this event, let us make the strong assumption that the checksum blocks are uniformly random. This differs from the actual model, since the blocks in the checksum follow a sum of uniform variables which, furthermore, depends on the value of the blocks. However, as we will see for concrete values, the gap between the real probability and this approximation is small for practical parameters.

Now, let us consider the above condition for one block $b'_i$ at one fixed index $0 < i \leq \ell$. Since $b_i^j,\ b'_i \sim \mathcal{U}([0,\ W-1])$ for all $0 \leq j \leq q$, conditioning the probability of this event produces [BH16]:

$$\begin{aligned}\mathbb{P}\left(\exists\ 0 \leq j \leq q\ \ b_i^j \leq b'_i\right) &= \sum_{x=0}^{W-1} \mathbb{P}\left(\exists\ 0 \leq j \leq q\ \ b_i^j \leq x \mid b'_i = x\right) \cdot \mathbb{P}\left(b'_i = x\right) \\ &= \sum_{x=0}^{W-1} \frac{1}{W} \cdot \left(1 - \left(\frac{W-(x+1)}{W}\right)^{q+1}\right).\end{aligned}$$

As every block is supposed uniform, the success probability is:

$$\mathbb{P}(\text{Success}) = \frac{1}{W^\ell}\left(\sum_{x=0}^{W-1}\left(1 - \left(\frac{W-(x+1)}{W}\right)^{q+1}\right)\right)^\ell.$$

#### 5.1.2.3 HORS(T)

Let the component be an implementation of the HORS scheme, as described in Section 3.3.1, or its improvement, HORST, which was explained in Section 3.4.6. The fault injection allows a straightforward reuse of the instance which directly degrades its security. In these schemes, the message digest is split into multiple blocks, but as opposed to W-OTS, the secret values are directly assigned to the value of these blocks. As before, the adversary requires to find another message such that the values for all the blocks have been revealed at least once in the faulty signatures.

Recovering the faulty digests in HORS can be difficult, as the attacker requires either the whole public key $Y_{\text{HORS}}$ or a way to verify a signature. In the unlikely scenario in which an attacker does not have access to any of these, and also cannot predict the effect of the induced fault, the corresponding faulty digests need to be guessed. This could add an additional layer of security that we will not cover here.

Assume the attacker has a way to recover the faulty digests, and let $(b_0^j, \ldots, b_{k-1}^j)$ be the result of splitting the valid digest for $i = 0$ and the faulty ones for $0 < i \leq q$ into blocks of length $\tau$. The attacker succeeds in forging a message $M' \neq M$ and its equivalent digest $D' = (b'_0, \ldots, b'_{k-1})$ when the following event occurs:

$$\text{Success} = \forall\ 0 \leq i < \ell\ \exists\ 0 \leq j \leq q\ \ b_i^j = b'_i.$$

Let us analyze the probability that a fixed block at index $i$ has been covered by the faulty signatures. As $b_i^j,\ b'_i \sim \mathcal{U}([0, 2^\tau - 1])$ for all $0 \leq j \leq q$, conditioning the probability of this event results in:

$$
\begin{aligned}
&\mathbb{P}\left(\exists\, 0 \le j \le q\ \exists\, 0 \le l < k\ \ b_l^j = b'_i\right) \\
&\qquad = \sum_{x=0}^{2^\tau - 1} \mathbb{P}\left(\exists\, 0 \le j \le q\ \exists\, 0 \le l < k\ \ b_l^j = x \mid b'_i = x\right) \cdot \mathbb{P}\left(b'_i = x\right) \\
&\qquad = \sum_{x=0}^{2^\tau - 1} \frac{1}{2^\tau} \cdot \left(1 - \left(1 - \frac{1}{2^\tau}\right)^{kq}\right) \\
&\qquad = \left(1 - \left(1 - \frac{1}{2^\tau}\right)^{kq}\right).
\end{aligned}
$$

#### 5.1.2.4 Success probabilities comparison

Figure 5.2 compares the different success rates from the above fault analyses. The values for the parameters have been chosen to achieve 128 bits of quantum security, as in SPHINCS-256. HORS performs the best against faults, as it has been designed to be reused. The security of LD-OTS and W-OTS, however, degrades exponentially fast with the number of faults.

In the case of W-OTS, we made a strong assumption on the distribution of the checksum blocks which could potentially provide a gap of success from real-case scenarios. To inspect this gap, we experimented the existential forgery with $10 \le q \le 100$ random signatures, and we recorded the number of trials for a forgery to succeed. The parameters chosen were the same as the previous comparison, and the experimentation were averaged from 10000 runs. Figure 5.3 compares the computed probability with the inverse of the number of trials. The average difference between the two curves is less than 2%.

### 5.1.3 Universal forgery on CMSS

A direct application of the existential forgeries from before is the creation of a universal forgery on a scheme that uses the tree chaining technique described in Section 3.4.3. When an error occurs during the construction of a non-top subtree in the signing procedure, a one-time signature instance is being reused with a different message. An attacker is therefore able to forge its own subtree and re-construct a whole part of the hypertree. The fault exploit is therefore apparent to a *tree grafting*: the CMSS hypertree is cut at a certain subtree, and the bottom is grafted with forged subtrees.

Assume a device runs a variant of a Merkle scheme using the tree chaining method (e.g., SPHINCS). Let $M \in \{0,1\}^*$ be a message and $\Sigma = (\sigma_{\mathrm{OTS}_0}, \mathrm{Auth}_0, \ldots, \sigma_{\mathrm{OTS}_{d-1}}, \mathrm{Auth}_{d-1})$ be the overall signature corresponding to $M$. Let $0 \le i < d-1$ be the index of a subtree in the path of the signature which is not located at the top of the hypertree. Now, suppose that the instance re-signs the same message $q$ times, but an error occurs during each construction of the subtree at index $i$. This produces $q$ different faulty signatures $(\hat{\Sigma}_0, \ldots, \hat{\Sigma}_{q-1})$ in which all the $\hat{\sigma}_{i+1}$ are different from $\sigma_{i+1}$. Using the secrets learnt from $\sigma_{i+1}$ and $\hat{\sigma}_{i+1}$, an attacker can forge a different subtree that is made valid using the existential forgery explained in 5.1.2. The created subtree can be then used to forge the

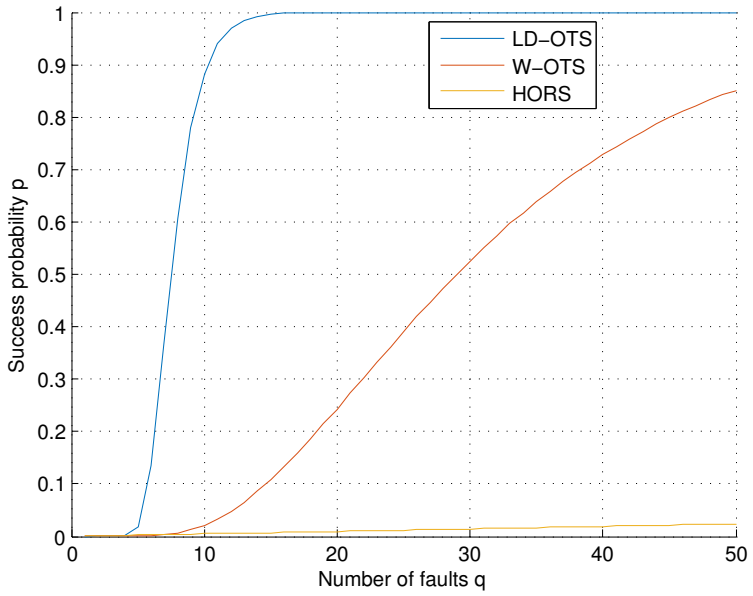



**Figure 5.2:** Comparison of success probability for LD-OTS ($n = 256$), W-OTS ($n = 256$, $w = 4$), and HORS ($n = 512$, $k = 32$, $\tau = 16$).

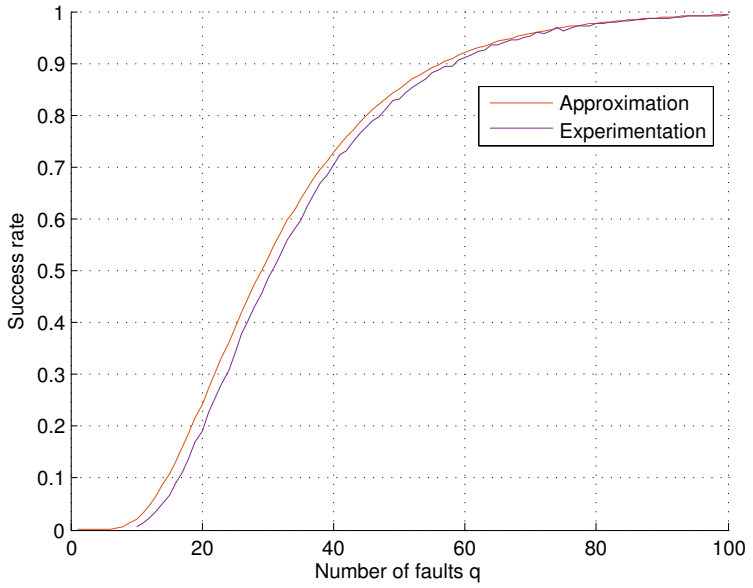



**Figure 5.3:** Comparison of W-OTS ($n = 256$, $w = 4$) probability of success versus the average success rate of 10000 experimentations.

bottom part of the hypertree which results in a universal forgery. The chance of success increases with the probability of performing the existential forgery.

Figure 5.4 illustrates the above attack. On the left, the highlighted subtree is targeted with FA when a message $M$ was being signed. Once the attacker collected enough faulty signatures, the subtree and all its branches are cut off the hypertree and replaced by a

forged subtree. The forged subtree allows the authentication of any other subtree which is then used to sign any other message $M'$.

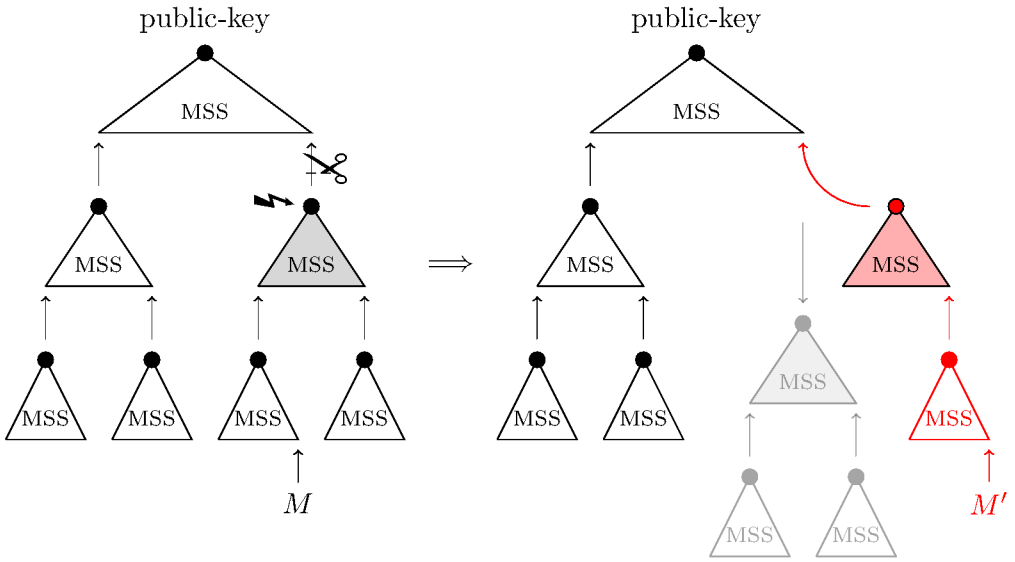



**Figure 5.4:** An illustration of the tree grafting attack against CMSS.

The attack works because the root of the subtree $i$ consists of the hash tree compression of the same OTS verification keys, so the OTS instance $(i+1)$ should always produce the same one-time signature $\sigma_{i+1}$. However, since a small change in any of the nodes which compose the subtree $i$ will necessarily propagate up to the root, a drastically different value will be signed. This small change can occur with any kind of fault, as a single bit flipped is sufficient to create a completely different root.

**Fault characteristics**

- **Number of faulty signatures** : $q$
- **Granularity** : a single to many bits
- **Modification** : random
- **Control** : loose
- **Target** : any value during a subtree construction

**Countermeasures**

As for the existential forgeries, a first protection against this attack would be to store and output every OTS. In this case, the occurrence of a fault would simply output a wrong signature, but not put the security of the scheme in jeopardy. This would obviously make the scheme impractical, however one could consider caching the OTS instances that are frequently used.

Also, while constructing the subtree, computing several times the hash of two nodes, or the subtree root, and comparing the results to each other can prevent the component to

sign a wrong message. The construction would only continue if all the results are the same, and resets or fails otherwise. Once again, computing a hash should be inexpensive, but here the time complexity would almost be multiplied by a constant. Moreover, the attack can still succeed but would require the attacker to bypass the comparisons, which is operable with non-invasive faults.

#### 5.1.3.1 Practical verification

In order to prove that such an attack presents a real risk to the security of hash-based schemes, this subsection details the procedure used to successfully attack a device that implements SPHINCS-256. The method used to inject the faults was the voltage glitching. The goal was to randomly corrupt the output hash of a tree node.

As in Chapter 4, the target device was an Arduino Due board [Ard17] that ran a SAM3X8E ARM Cortex-M3 implementation of SPHINCS-256, as described in Section 3.4. The specific components on the board are referenced according to the official Arduino Due schematic [Mar15] and the official Atmel ARM-based SAM3X8E datasheet [Atm15].

The attack scenario considers the SPHINCS-256 implementation with fixed parameters to sign the same message. Since the whole signing procedure is quite long, the entire calculation was processed only once to create a valid SPHINCS signature $\Sigma$. Next, the construction of the first subtree was isolated with the appropriate root to sign and address. We used serial communication to communicate with the board. The signing procedure waited for a signal before processing the partial signature which was then output. The code was modified such that the computation of an arbitrary tree node would toggle a GPIO pin just before the call of the compression hash function $g$.

The fault injection technique used was the *voltage glitching* which temporarily interrupts the supply line of the circuit [Bar+12]. The attack is low-cost, requires a high precision in time but gives no control on the bit modification. It demands moderate technical skill to be performed. Voltage glitches are known to induce faults, from a single bit flipped to the corruption of a whole register, by modifying propagation times and reference voltages.

In order to glitch the Arduino Due board, a direct access to the core power (`VDDCORE`) was required. This operation is tricky, as it necessitates to create a shortcut from `VDDOUT` to `VDDDPLL` without supplying `VDDCORE`. This was done by lifting the `VDDOUT` pin from the microcontroller and soldering a wire on it (red wire). We then cut `VDDPLL` from `VDDCORE` by replacing the ferrite bead on the path (L5 MH2029-300Y) with another wire (orange wire). Finally, we soldered a wire (blue wire) to the now decoupled `VDDCORE` to obtain our access to it. Because the capacitors on the circuit would absorbe short power outages, removing all the ones in the path of `VDDOUT` (C11 to C17) was necessary to obtain results. A pulse generator was used to externally supply the core power by the blue wire, so glitches could be injected. Pictures of the modified board is shown in Figure 5.5.

We choose to place the trigger right before the computation of the node $\nu_3[1]$ in the first subtree, so the fault propagates up to the subtree root computation. The pulse generator would fix the power supply at a certain voltage for a few nanoseconds after it receives the trigger. A frequent corruption occurred with the following parameters:

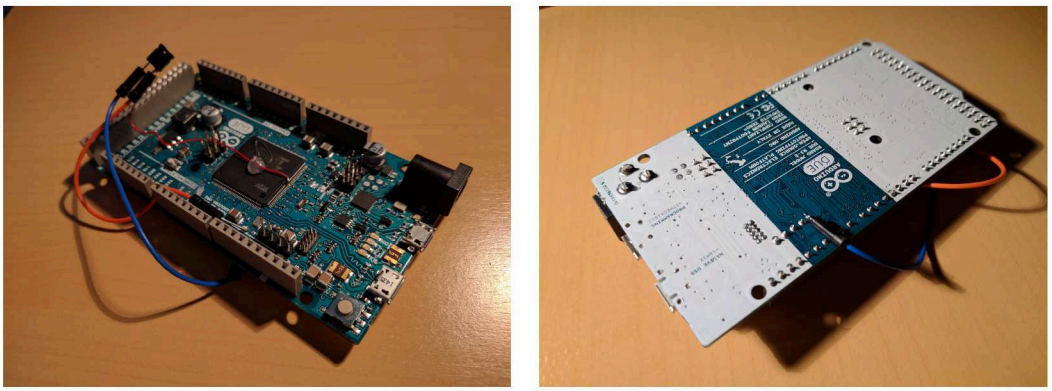


**Figure 5.5:** Arduino Due board setup for VDDCORE glitching.

### Voltage glitch parameters

- **Delay** : 195 ns
- **Width** : 37 ns
- **Voltage** : 0.85 V
- **Depth** : -3.1 V

Figure 5.6 shows the short power outage induced when the above parameters were used. The spike from the curve below reads the GPIO pin responsible for triggering the glitch, which is characterized by a spike in the `VDDOUT` power trace above.

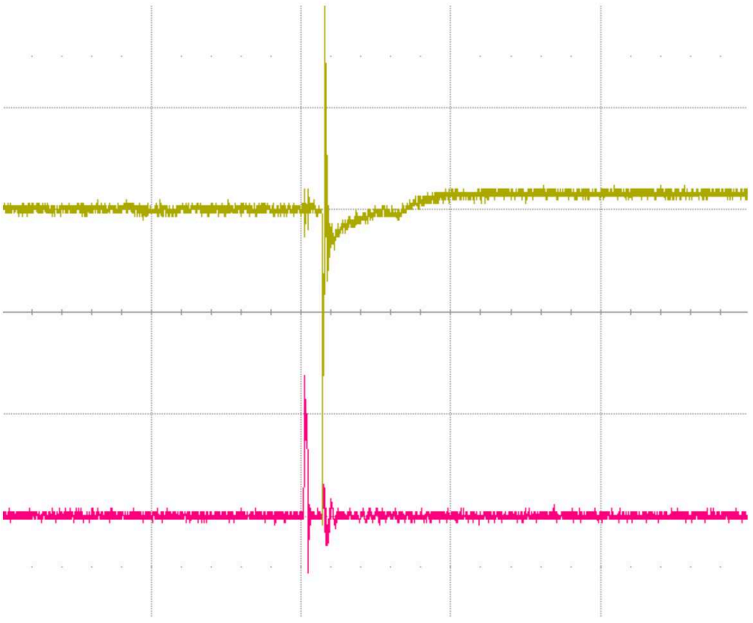


**Figure 5.6:** Glitch injection in `VDDCORE` (yellow) after the toggle (magenta) was received.

The process was repeated 10000 times which allowed us to collect about 100 faulty signatures. Interestingly, a few of them were the same, which led to a total of only $q = 85$

different faulty signatures.

In order to exploit the collected faulty signatures to produce the universal forgery, post-processing must be applied. The procedure first consists of finding the W-OTS$^+$ public-key using the valid signature $\sigma$. This is done by computing the valid subtree root and applying the W-OTS$^+$ verification process (Algorithm 3.1.4) in such a way that the procedure output the presumed public key $Y$. Then, a table is created such that, for each block, the lowest block value known is mapped to their corresponding point in their hash chain. This table can start with the values inside the valid signature $\sigma$. The secret values identification in $\hat{\sigma}_i$ for $0 \leq i < q$ consists of applying the chaining function $c^i(x, r_{\text{W-OTS}^+})$ to them with all different $i$ until the function outputs the value of the public-key. If one value is lower than the corresponding one in the table, then the table is updated with it and its value.

Once the table is completed, the W-OTS$^+$ signing procedure (Algorithm 3.1.3) is ready to be attempted. This process will try to sign a forged subtree root if the values inside of the table allow it. To construct a subtree, we decided to find our own $\text{SK}_1$ to create the W-OTS$^+$ at the leaves as in SPHINCS, so recovering the instances is more convenient. Therefore, the algorithm simply chooses $\text{SK}'_1$ at random, processes the $\text{SEED}'_A$ at the same addresses as the valid signature, computes the subtree root, and finally tries to sign it. If the attempt fails, the above process is repeated with a different $\text{SK}'_1$. Otherwise, the resulting $\sigma'$ is a valid signature for the subtree created with $\text{SK}'_1$.

The rest of the attack consists of using an instance of W-OTS$^+$ to sign the root of a forged HORST which corresponds to the signature of a message of our choice $M'$. This is simply done by signing $M'$ using a random HORST instance whose root is signed with the W-OTS$^+$ instance located at the same address as the subtree attacked. The resulting two signatures are then attached to the forged signature $\sigma'$, and then to the rest of the valid SPHINCS signature $\Sigma$.

**Results**

Using only $q = 20$ faulty signatures, we managed to forge our own subtree with 18 attempts. The attack allowed the valid signature forgery of a message of our choice. All the details, along with concrete values for parameters and signatures, can be found at `https://github.com/AymericGenet/SPHINCS-arduinodue` under the `faults` folder.

The number of attempts is directly linked to the success of the W-OTS$^+$ existential forgery from Section 5.1.2.2. Figure 5.7 shows the comparison of the theoretical probability of success versus the success rate of this attack when the number of faulty signatures increases.

## 5.2 Invasive attacks

Invasive attacks describe the scenarios in which the adversary has total physical access to the silicon die from the device chip, in addition to the ability to target individual circuits in a very precise manner [Bar+12]. With these capabilities, the faults induced can affect entire values by fixing a specific value to them, but often leave permanent damage to the device. An example is the projection of a precise laser beam on logic gates that cause many bits from intermediate values to be canceled out.

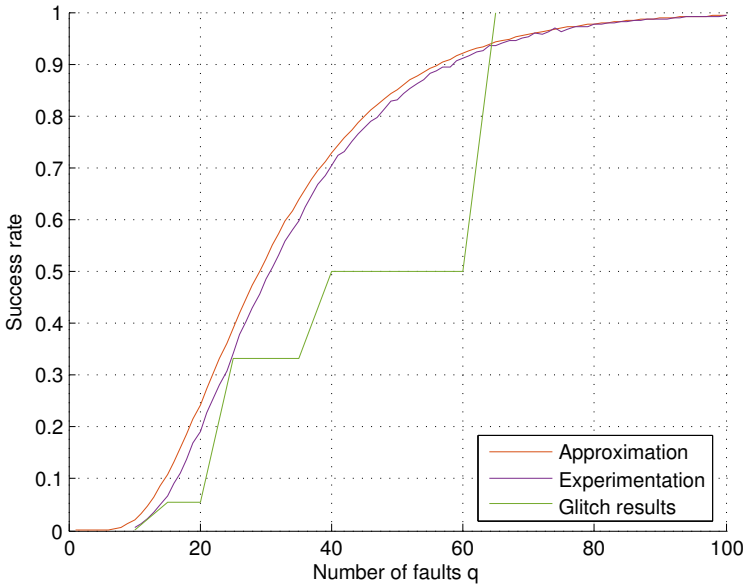



**Figure 5.7:** Success rate of the SPHINCS-256 universal forgery ($n = 256$, $w = 4$) by increasing the number of faults $q$, compared to the theoretical and experimented probabilities of success.

### 5.2.1 Key recovery on W-OTS($^+$)

Even if a W-OTS or W-OTS$^+$ instance, as described in Section 3.1.2 and Section 3.4.5, is only used once to sign a unique message, the component stays vulnerable to a more powerful kind of FA. In case an attacker is able to force the output of the block splitting subroutine to be stuck at zero, the component is forced to reveal all its secret values, resulting in a total break.

Suppose that a device runs any authentication scheme which uses instances of any variant of W-OTS only once for a unique message (e.g., original MSS). If an attacker can cause the blocks from the message digest and the checksum to be stuck at zero, the signing procedure produces a single $\bar{\sigma}_{\text{W-OTS}}$ that contains the entire secret key vector $X_{\text{W-OTS}}$ from the instance. The attack also works if the attacker directly targets both the digest and the checksum computations.

The reason for this attack comes from the simple fact that the values revealed correspond to the points in the hash chains which all start from a different secret value. Forcing all the blocks to be zero causes the device to output the starting points of all the chains. This obviously permits a total reconstruction of any point in the chains which further produces a key recovery.

**Fault characteristics**

- **Number of faulty signatures** : 1
- **Granularity** : entire values
- **Modification** : stuck-at 0
- **Control** : extremely precise
- **Target** : the message digest computation and the checksum

While being very effective, this attack requires the aptitude of canceling a whole output of a subroutine. This supposes that the adversary has an extremely strong capability on the device, which is rarely the case or has a high cost. However, even if the attack is unlikely, it should not be ignored, especially if the data signed is highly sensitive.

**Countermeasures**

Because of the extreme capability of the adversary, countermeasures against such an attack are difficult to develop. The classic ones involve error detection codes, such as performing the signature twice, hardware shielding, and multi-area checking. These however only make the job of the adversary harder, as they can never totally protect the device.

Interestingly, if fixing values to one happens to be harder than fixing to zero, one could consider reversing the order of the chains. This is the case, for instance, for laser beam injection, which can usually cancel out a bit, but not set it. In this case, the secret values would only be output if the block values are all one, instead of zero.

# Chapter 6

# Conclusion

This thesis has examined the different vulnerabilities of hash-based cryptography with power analysis and fault injection attacks. In summary, the existence of a leakage in the reference implementation of BLAKE-256 was proved through DPA. Although it did not allow us to recover the secret key, many critical faults attacks on various hash-based schemes were discovered, which led from existential forgeries to universal forgeries. The practicality of the universal forgery against SPHINCS-256 was also demonstrated using voltage glitching.

In the future, a further analysis of the BLAKE-256 leakage through DPA in the context of SPHINCS to determine whether key recovery can be performed would be interesting to conduct. Also, the current research could be extended with other types of hardware attacks, such as template attacks, or a synergy between faults and side-channel attacks. Finally, the subject can be completed by attacking the recent hash-based zero-knowledge proofs of Fish and Picnic with hardware attacks.

In conclusion, the thesis gave a good insight into the risks of implementing hash-based schemes on hardware devices. The schemes are especially robust against power analysis, since the recovery of secret information is reduced to the deduction of input from hash function execution. Moreover, if these hash functions are known to leak an unacceptable amount of information, they can easily be replaced by more resistant ones. However, the design itself of hash-based schemes has been proved to be extremely vulnerable to the presence of faults. As the adversary does not need powerful capabilities to seriously compromise the security of cryptosystems, this heavily discourages unprotected embedded implementations of hash-based schemes. Countermeasures to mitigate these attacks were discussed, but the risk can never be totally eliminated.

Nevertheless, the knowledge of such flaws permits a better understanding of the security in hash-based cryptography, which is an important process in the development of safer schemes. Improving the security of schemes is an essential concern, as some schemes will eventually be used to secure personal information. As a result, this thesis is a direct contribution to the safety of the human society, and therefore, to a better world.

# Bibliography


[Lam79] Leslie Lamport. *Constructing digital signatures from a one-way function.* Tech. rep. Technical Report CSL-98, SRI International Palo Alto, 1979.

[Gol86] Oded Goldreich. "Two Remarks Concerning the Goldwasser-Micali-Rivest Signature Scheme". In: *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings.* Ed. by Andrew M. Odlyzko. Vol. 263. Lecture Notes in Computer Science. Springer, 1986, pp. 104–110. DOI: 10.1007/3-540-47721-7_8. URL: http://dx.doi.org/10.1007/3-540-47721-7_8.

[Mer89] Ralph C. Merkle. "A Certified Digital Signature". In: *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings.* Ed. by Gilles Brassard. Vol. 435. Lecture Notes in Computer Science. Springer, 1989, pp. 218–238. ISBN: 3-540-97317-6. DOI: 10.1007/0-387-34805-0_21. URL: http://dx.doi.org/10.1007/0-387-34805-0_21.

[Rom90] John Rompel. "One-Way Functions are Necessary and Sufficient for Secure Signatures". In: *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA.* Ed. by Harriet Ortiz. ACM, 1990, pp. 387–394. ISBN: 0-89791-361-2. DOI: 10.1145/100216.100269. URL: http://doi.acm.org/10.1145/100216.100269.

[Kah96] D. Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet.* Scribner, 1996. ISBN: 9781439103555. URL: https://books.google.ch/books?id=SEH_rHkgaogC.

[MOV96] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996. ISBN: 0-8493-8523-7.

[BR97] Mihir Bellare and Phillip Rogaway. "Collision-Resistant Hashing: Towards Making UOWHFs Practical". In: *IACR Cryptology ePrint Archive* 1997 (1997), p. 9. URL: http://eprint.iacr.org/1997/009.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)". In: *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding.* Ed. by Walter Fumy. Vol. 1233. Lecture Notes in Computer Science. Springer, 1997, pp. 37–51. ISBN: 3-540-62975-0. DOI: 10.1007/3-540-69053-0_4. URL: https://doi.org/10.1007/3-540-69053-0_4.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 388–397. ISBN: 3-540-66347-9. DOI: 10.1007/3-540-48405-1_25. URL: https://doi.org/10.1007/3-540-48405-1_25.

[MDS99] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. "Power Analysis Attacks of Modular Exponentiation in Smartcards". In: *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*. Ed. by Çetin Kaya Koç and Christof Paar. Vol. 1717. Lecture Notes in Computer Science. Springer, 1999, pp. 144–157. ISBN: 3-540-66646-X. DOI: 10.1007/3-540-48059-5_14. URL: https://doi.org/10.1007/3-540-48059-5_14.

[Sho99] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM Review* 41.2 (1999), pp. 303–332. DOI: 10.1137/S0036144598347011. URL: https://doi.org/10.1137/S0036144598347011.

[AJ01] Isabelle Attali and Thomas P. Jensen, eds. *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*. Vol. 2140. Lecture Notes in Computer Science. Springer, 2001. ISBN: 3-540-42610-8. DOI: 10.1007/3-540-45418-7. URL: https://doi.org/10.1007/3-540-45418-7.

[Gol01] Oded Goldreich. *The Foundations of Cryptography - Volume 1, Basic Techniques*. Cambridge University Press, 2001. ISBN: 0-521-79172-3.

[Per01] Adrian Perrig. "The BiBa one-time signature and broadcast authentication protocol". In: *CCS 2001, Proceedings of the 8th ACM Conference on Computer and Communications Security, Philadelphia, Pennsylvania, USA, November 6-8, 2001*. Ed. by Michael K. Reiter and Pierangela Samarati. ACM, 2001, pp. 28–37. ISBN: 1-58113-385-5. DOI: 10.1145/501983.501988. URL: http://doi.acm.org/10.1145/501983.501988.

[RR02] Leonid Reyzin and Natan Reyzin. "Better than BiBa: Short One-time Signatures with Fast Signing and Verifying". In: *IACR Cryptology ePrint Archive* 2002 (2002), p. 14. URL: http://eprint.iacr.org/2002/014.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. "Correlation Power Analysis with a Leakage Model". In: *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*. Ed. by Marc Joye and Jean-Jacques Quisquater. Vol. 3156. Lecture Notes in Computer Science. Springer, 2004, pp. 16–29. ISBN: 3-540-22666-4. DOI: 10.1007/978-3-540-28632-5_2. URL: https://doi.org/10.1007/978-3-540-28632-5_2.

[Gol04] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Basic Applications. Cambridge University Press, 2004. ISBN: 9780521830843. URL: https://books.google.ch/books?id=tfzM9d_jnxwC.

[Nat04] National Institute of Standards and Technology. *FIPS PUB 180-2 (with Change Notice 1): Secure Hash Standards*. Gaithersburg, MD, USA: National Institute for Standards and Technology, Feb. 2004. URL: http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf.

[Gar05] Luis Carlos Coronado García. "Provably secure and practical signature schemes". PhD thesis. Darmstadt University of Technology, Germany, 2005. URL: http://elib.tu-darmstadt.de/diss/000642.

[NSW05] Dalit Naor, Amir Shenhav, and Avishai Wool. "One-Time Signatures Revisited: Have They Become Practical?" In: *IACR Cryptology ePrint Archive* 2005 (2005), p. 442. URL: http://eprint.iacr.org/2005/442.

[Buc+06] Johannes A. Buchmann, Luis Carlos Coronado García, Erik Dahmen, Martin Döring, and Elena Klintsevich. "CMSS - An Improved Merkle Signature Scheme". In: *IACR Cryptology ePrint Archive* 2006 (2006), p. 320. URL: http://eprint.iacr.org/2006/320.

[Ber08] Daniel J. Bernstein. *The ChaCha family of stream ciphers*. 2008. URL: https://cr.yp.to/chacha.html (visited on 08/18/2017).

[Dah+08] Erik Dahmen, Katsuyuki Okeya, Tsuyoshi Takagi, and Camille Vuillaume. "Digital Signatures Out of Second-Preimage Resistant Hash Functions". In: *Post-Quantum Cryptography, Second International Workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17-19, 2008, Proceedings*. Ed. by Johannes A. Buchmann and Jintai Ding. Vol. 5299. Lecture Notes in Computer Science. Springer, 2008, pp. 109–123. ISBN: 978-3-540-88402-6. DOI: 10.1007/978-3-540-88403-3_8. URL: http://dx.doi.org/10.1007/978-3-540-88403-3_8.

[SGV08] François-Xavier Standaert, Benedikt Gierlichs, and Ingrid Verbauwhede. "Partition vs. Comparison Side-Channel Distinguishers: An Empirical Evaluation of Statistical Tests for Univariate Side-Channel Attacks against Two Unprotected CMOS Devices". In: *Information Security and Cryptology - ICISC 2008, 11th International Conference, Seoul, Korea, December 3-5, 2008, Revised Selected Papers*. Ed. by Pil Joong Lee and Jung Hee Cheon. Vol. 5461. Lecture Notes in Computer Science. Springer, 2008, pp. 253–267. ISBN: 978-3-642-00729-3. DOI: 10.1007/978-3-642-00730-9_16. URL: https://doi.org/10.1007/978-3-642-00730-9_16.

[Ber09] Daniel J Bernstein. "Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete". In: *SHARCS'09 Special-purpose Hardware for Attacking Cryptographic Systems* (Sept. 2009), p. 105.

[BDS09] Johannes Buchmann, Erik Dahmen, and Michael Szydlo. "Hash-based Digital Signature Schemes". In: *Post-Quantum Cryptography*. Ed. by Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 35–93. ISBN: 978-3-540-88702-7. DOI: 10.1007/978-3-540-88702-7_3. URL: http://dx.doi.org/10.1007/978-3-540-88702-7_3.

[Gie+09] Benedikt Gierlichs, Elke De Mulder, Bart Preneel, and Ingrid Verbauwhede. "Empirical comparison of side channel analysis distinguishers on DES in hardware". In: *Circuit Theory and Design, 2009. ECCTD 2009. European Conference on*. IEEE. 2009, pp. 391–394.

[Hna+10] William Hnath, Jordan Pettengill, Alan Chen, Brendon Chetwynd, Evan Custodio, and Sam Ianni. "Differential Power Analysis Side-Channel Attacks in Cryptography". In: 2010.

[KH10] Jongsung Kim and Seokhie Hong. "Side-Channel Attack Using Meet-in-the-Middle Technique". In: *Comput. J.* 53.7 (2010), pp. 934–938. DOI: 10.1093/comjnl/bxp054. URL: https://doi.org/10.1093/comjnl/bxp054.

[Buc+11] Johannes A. Buchmann, Erik Dahmen, Sarah Ereth, Andreas Hülsing, and Markus Rückert. "On the Security of the Winternitz One-Time Signature Scheme". In: *IACR Cryptology ePrint Archive* 2011 (2011), p. 191. URL: http://eprint.iacr.org/2011/191.

[Koc+11] Paul C. Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. "Introduction to differential power analysis". In: *J. Cryptographic Engineering* 1.1 (2011), pp. 5–27. DOI: 10.1007/s13389-011-0006-y. URL: https://doi.org/10.1007/s13389-011-0006-y.

[Wik11] Wikimedia. *File:Power attack.png — Wikimedia Commons, the free media repository*. 2011. URL: https://commons.wikimedia.org/w/index.php?title=File:Power_attack.png&oldid=54038779 (visited on 08/18/2017).

[Bar+12] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. "Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures". In: *Proceedings of the IEEE* 100.11 (2012), pp. 3056–3076. DOI: 10.1109/JPROC.2012.2188769. URL: https://doi.org/10.1109/JPROC.2012.2188769.

[Hül13] Andreas Hülsing. "W-OTS+ - Shorter Signatures for Hash-Based Signature Schemes". In: *Progress in Cryptology - AFRICACRYPT 2013, 6th International Conference on Cryptology in Africa, Cairo, Egypt, June 22-24, 2013. Proceedings*. Ed. by Amr Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien. Vol. 7918. Lecture Notes in Computer Science. Springer, 2013, pp. 173–188. ISBN: 978-3-642-38552-0. DOI: 10.1007/978-3-642-38553-7_10. URL: http://dx.doi.org/10.1007/978-3-642-38553-7_10.

[Aum14] Jean-Philippe Aumasson. *BLAKE*. 2014. URL: https://github.com/veorq/BLAKE/ (visited on 08/18/2017).

[Aum+14] Jean-Philippe Aumasson, Willi Meier, Raphael C.-W. Phan, and Luca Henzen. *The Hash Function BLAKE*. Information Security and Cryptography. Springer, 2014. ISBN: 978-3-662-44756-7. DOI: 10.1007/978-3-662-44757-4. URL: https://doi.org/10.1007/978-3-662-44757-4.

[Bal14] Josep Balasch. "Implementation Aspects of Security and Privacy in Embedded Design". Ingrid Verbauwhede and Bart Jacobs (promotors). PhD thesis. KU Leuven and Radboud University Nijmegen, 2014.

[Ber+14] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Peter Schwabe, and Zooko Wilcox-O'Hearn. "SPHINCS: practical stateless hash-based signatures". In: *IACR Cryptology ePrint Archive* 2014 (2014), p. 795. URL: http://eprint.iacr.org/2014/795.

[Atm15] Atmel. *SMART ARM-based MCU Datasheet*. Mar. 2015. URL: http://www.atmel.com/Images/Atmel-11057-32-bit-Cortex-M3-Microcontroller-SAM3X-SAM3A_Datasheet.pdf.

[HRS15] Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. "ARMed SPHINCS - Computing a 41KB signature in 16KB of RAM". In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 1042. URL: http://eprint.iacr.org/2015/1042.

[Mar15] Gianluca Martino. *Arduino Due Schematic*. Jan. 2015. URL: http://download.arduino.org/products/DUE/Arduino_DUE-V02e-SCH.pdf.

[BH16] Leon Groot Bruinderink and Andreas Hülsing. ""Oops, I did it again" - Security of One-Time Signatures under Two-Message Attacks". In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 1042. URL: http://eprint.iacr.org/2016/1042.

[Hod+16] Philip Hodgers, Francesco Regazzoni, Richard Gilmore, Ciara Moore, and Tobias Oder. *State-of-the-Art in Physical Side-Channel Attacks and Resistant Technologies*. Version 1. Feb. 2016. URL: https://www.safecrypto.eu/outcomes/deliverables/.

[Ard17] Arduino. *Arduino Due*. 2017. URL: https://store.arduino.cc/usa/arduino-due (visited on 08/18/2017).

[Cha+17] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. "Post-Quantum Zero-Knowledge and Signatures from Symmetric-Key Primitives". In: *IACR Cryptology ePrint Archive* 2017 (2017), p. 279. URL: http://eprint.iacr.org/2017/279.

[Ste+17] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. "The first collision for full SHA-1". In: *IACR Cryptology ePrint Archive* 2017 (2017), p. 190. URL: http://eprint.iacr.org/2017/190.